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 поиска пути в графах

Студентка гр. 8304 _____ Николаева М. А.

Преподаватель _____ Размочаева Н. В.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с алгоритмами поиска пути в графах на примере жадного алгоритмом на графе и алгоритма A*. Научиться применять данные алгоритмы для решения задач, а также оценивать временную сложность алгоритмов.

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.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

2) Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе именуется целыми числами (в т. ч. отрицательными), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Описание жадного алгоритма.

На первом шаге рассматривается стартовая вершина. На каждом последующем шаге будут рассматриваться вершины, в которые возможно попасть напрямую из текущей вершины. Далее среди этих вершин выбирается вершина, рассто-

яние до которой от текущей является наименьшим. Эта вершина помечается рассмотренной и выбирается текущей. Предыдущий шаг повторяется. Если из текущей вершины не существует пути в еще не рассмотренные до этого вершины, то происходит возврат в вершину, из которой был совершен переход в текущую.

Алгоритм заканчивает работу в двух случаях:

1) Когда текущей вершиной становится конечная (путь найден).
2) Когда все вершины были рассмотрены.

Описание алгоритма A*.

Стартовая вершина помечается заносится в «открытую». Пока существуют «открытые» вершины:

1) В качестве текущей выбирается открытая, с наименьшим значением f (полной стоимостью).
2) Если текущая вершина является конечной, то алгоритм заканчивает свою работу (путь существует).
3) Переносим текущую вершину в «закрытые».

Для каждого ребенка текущей вершины, который не является «закрытым» находим длину пути.

Если вершина еще не закрыта или рассчитанная функция пути меньше функции, рассчитанной для этой вершины ранее, то присваиваем значение рассчитанной функции этой вершине. Запоминаем вершину, из которой мы пришли в ребенка для того, чтобы потом восстановить путь.

Если открытых вершин не осталось, алгоритм заканчивает работу (пути не существует).

Анализ алгоритмов.

Оценим временную сложность алгоритмов: $O(E + V^2) = O(V^2)$, где V – кол-во вершин, E – кол-во ребер. В худшем случае для поиска пути потребуется обход всех ребер и вершин.

Оценим затраты на хранение: $O(E + V)$, где V – кол-во вершин, E – кол-во ребер, затрачивается на хранение графа. Для хранения в программе используется V – вершин и E – указателей на смежные вершины.

Описание функций и структур данных.

```
struct Vertex
{
	Vertex(int name) : name(name) { }

	double g;
	double f;
	int name;
	std::vector <Child*> children;
};
```

Структура для хранения графа. Содержит поля: имя вершины графа, результат функции пути g, результат полной функции f с учетом эвристики h, массив структур Child, где Child – это структура данных для хранения вершины и веса пути до вершины из текущей.

```
struct Child
{
	Child(Vertex* vertex, double pathLen) : vertex(vertex), pathLen(pathLen) { }

	Vertex* vertex;
	double pathLen;
};
```

Содержит указатель на вершину и длину пути до вершины.

```
int h(Vertex* vertex1, Vertex* vertex2)
```

Функция для вычисления эвристической функции (в данном решении – расстояние между символами в таблице ASCII). Принимает две вершины и возвращает значение эвристической функции.

`bool aStar(Vertex* start, Vertex* end)` – функция алгоритма A*. Примает начальную и конечную вершину. Возвращаемое значение bool.

Тестирование программы.

Результаты тестирование разработанной программы представлены в табл. 1.

Таблица 1 – Результаты тестирование программы

Ввод	Вывод
1 3 1 2 10 2 3 10	Path between start vertex and end vertex found: 1->2->3
1 5 1 2 1 2 4 10 4 5 1 1 3 2 3 5 20	Path between start vertex and end vertex found: 1->2->4->5
1 5 1 2 3 2 3 1 3 4 1 1 4 5 5 6 1	Path between start vertex and end vertex found: 1->4->5
-5 3 -5 -2 1 -2 0 1 0 3 100 1 3 2 -5 1 200 -5 0 10 0 1 5	Path between start vertex and end vertex found: -5->-2->0->1->3

Выводы.

В ходе выполнения лабораторной работы были реализованы алгоритмы поиска пути в графе, дана оценка времени работы алгоритмов и требуемой памяти.

Приложение.
Код работы.

main.cpp:

```
#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include <stack>
#include <set>


struct Vertex;
struct Child;

bool aStar(Vertex* start, Vertex* end);
Vertex* vertexWithMinF(std::set<Vertex*> open);
int h(Vertex* vertex1, Vertex* vertex2);


struct Vertex
{
	Vertex(int name) : name(name) { }

	double g;
	double f;
	int name;
	std::vector <Child*> children;
};


struct Child
{
	Child(Vertex* vertex, double pathLen) : vertex(vertex), pathLen(pathLen) { }

	Vertex* vertex;
	double pathLen;
};


int h(Vertex* vertex1, Vertex* vertex2) {
	return abs(abs(vertex1->name) - abs(vertex2->name));
}


Vertex* vertexWithMinF(std::set<Vertex*> open) {
	Vertex* vertexWithMinF = nullptr;

	for (auto i : open) {
		if (!vertexWithMinF) {
			vertexWithMinF = i;
		}
		else if (vertexWithMinF->f > i->f) {
			vertexWithMinF = i;
		}
		else if (vertexWithMinF->f == i->f) {
			if (vertexWithMinF->name < i->name) {
				vertexWithMinF = i;
			}
```

```
			}
		}

		return vertexWithMinF;
	}



	bool aStar(Vertex* start, Vertex* end) {
		std::set<Vertex*> open;
		std::set<Vertex*> closed;
		std::map<Vertex*, Vertex*> from;

		start->g = 0;
		start->f = start->g + h(start, end);
		open.insert(start);


		while (!open.empty()) {
			Vertex* current = vertexWithMinF(open);

			if (current == end) {
				std::cout << "Path between start vertex and end vertex found:\n";
				std::stack<int> vertexNames;

				while (from.find(current) != from.end()) {
					vertexNames.push(current->name);
					current = from[current];
				}
				vertexNames.push(start->name);

				std::cout << vertexNames.top();
				vertexNames.pop();
				while (!vertexNames.empty()) {
					std::cout << "->" << vertexNames.top();
					vertexNames.pop();
				}
				std::cout << "\n";
				return true;
			}

			closed.insert(current);
			open.erase(open.find(current));

			for (auto child : current->children) {
				if (closed.find(current) != closed.end()) {
					double tmpChildG = current->g + child->pathLen;

					if (child->vertex->g > tmpChildG || closed.find(child-
>vertex) == closed.end())  {
						from[child->vertex] = current;
						child->vertex->g = tmpChildG;
						child->vertex->f = child->vertex->g + h(child->ver-
tex, end);

						if (open.find(child->vertex) == open.end()) {
							open.insert(child->vertex);
						}
					}
				}
			}
```

```
	}

	return false;
}


int main()
{
	int startVertex = 0;
	int endVertex = 0;

	std::cin >> startVertex >> endVertex;
	Vertex* start = new Vertex(startVertex);
	Vertex* end = new Vertex(endVertex);

	std::map<int, Vertex*> connectionMap;
	connectionMap[start->name] = start;
	connectionMap[end->name] = end;

	int firstVertex = 0;
	int secondVertex = 0;
	double length = 0;

	while (!std::cin.eof()) {
		std::cin >> firstVertex >> secondVertex >> length;

		if (connectionMap.find(firstVertex) != connectionMap.end() &&
			connectionMap.find(secondVertex) != connectionMap.end()) {
			Child* child = new Child(connectionMap[secondVertex], length);
			connectionMap[firstVertex]->children.push_back(child);
		}
		else if (connectionMap.find(firstVertex) != connectionMap.end()) {
			Vertex* vertex = new Vertex(secondVertex);
			connectionMap[secondVertex] = vertex;

			Child* child = new Child(connectionMap[secondVertex], length);
			connectionMap[firstVertex]->children.push_back(child);
		}
		else if (connectionMap.find(secondVertex) != connectionMap.end()){
			Vertex* vertex = new Vertex(firstVertex);
			connectionMap[firstVertex] = vertex;

			Child* child = new Child(connectionMap[secondVertex], length);
			vertex->children.push_back(child);
		}
		else {
			Vertex* vertex = new Vertex(firstVertex);
			connectionMap[firstVertex] = vertex;

			Vertex* vertex2 = new Vertex(secondVertex);
			connectionMap[secondVertex] = vertex2;

			Child* child = new Child(connectionMap[secondVertex], length);
			connectionMap[firstVertex]->children.push_back(child);
		}
	}


	if (!aStar(start, end)) {
		std::cout << "No path found between start and end vertex\n";
	}
```

```
        return 0;
}
```