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. 0382 _____ Бочаров Г.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Изучить принцип работы алгоритма A* и жадного алгоритма. Решить с их помощью задачу нахождения пути в графе.

Задание

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.

Алгоритм A*:

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.

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.

Индивидуализация:

Вар. 4:

Модификация A* с двумя финишами (требуется найти путь до любого из двух).

Выполнение работы.

Исходный код программы представлен в приложении А.

Жадный алгоритм:

Сначала программа считывает 2 символа, начальная и конечная вершины.

С помощью функции read_graph считывается сам граф. Для хранения графа используется словарь, где ключом является название вершины, а значением множество ребер (структура Link), которые выходят из данной вершины. Ребра в множестве сортируются в порядке возрастания их длин.

Структура ребра (Link) хранит в себе название вершину в которую оно ведет и длину ребра. Для структуры переопределен оператор «меньше», чтобы ребра можно было упорядочить.

После считывания графа функция find находит путь между указанными вершинами графа. Функция find работает рекурсивно, по следующему алгоритму:

Берется последняя посещенная вершина (в самом начале берется стартовая). После чего для данной вершины находится самое короткое ребро, выходящее из нее и не ведущая в уже посещенные вершины. После посещается вершина в которую ведет данное ребро. Посещенная вершина вносится в множество посещенных вершин visited, чтобы не посещать ее еще раз. Далее для посещенной вершины рекурсивно вызывается функция find.

В случае нахождения пути до цели, поиск других путей прекращается.

Промежуточные решения хранятся в списке Path. При посещении, вершина добавляется в конец списка. Если выбранный путь зашел в тупик, последняя посещенная вершина удаляется из списка.

В худшем случае алгорим обойдет все ребра и вершины по одному разу и сложность будет $O(|V| + |E|$.

Алгоритм A* с двумя финишами:

Способ хранения и считывания графа, аналогичны способам используемым в жадном алгоритме, с той лишь разницей, что считывается второй финиш.

Алгоритм A* реализован следующим образом:

Для хранения стоимостей перемещений создан словарь cost_visited, где ключом является название вершины а значением расстояние от начального пункта до данной вершины.

Для хранения пути и посещенных вершин создан словарь visited, где каждой вершине соответствует вершина из которой мы прибыли в данную.

Для выбора оптимальной для следующего посещения вершины была создана очередь с приоритетом, в которую добавляются вершины(Vertex).

Структура Vertex хранит название вершины и оценочную стоимость пути к ней (расстояние от старта + эвристическая оценка).

Первым элементом очереди с приоритетом всегда будет вершина с наименьшей оценочной стоимость перемещения.

При посещении алгоритмом очередной вершины, все достижимые из данной вершины добавляются в очередь с приоритетом. После чего выбирается наиболее выгодная для посещения вершина из очереди. При этом вершина может быть посещена несколько раз, если новый путь, ведущий в нее будет более выгодным.

Процесс продолжается до тех пор пока в очереди не останется вершин для посещения или до тех пор пока стоимость посещения очередной вершины (самой выгодной из оставшихся) не будет превосходить длины уже найденого пути из начальной точки в конечную, который и будет кратчайшим.

В случае двух финишей, эвристическая функция считает минимальное расстояние между символом стартовой вершины и одним из финишей.

Сложность по времени алгоритма A* зависит от используемой эвристической функции. В худшем случае, число вершин, посещаемое алгоритмом, растёт экспоненциально относительно длины оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$|h(x) — h^*(x)| <= O(\log h^*(x))$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

Используема в данной работе эвристика далека от идеальной, так как расстояние между символами, не несет никакой информации о реальном расположении вершин в графе.

Результаты тестирования представлены в Таблице Б.1 (Приложение Б.)

Выводы.

Были изучены и исследованы жадный алгоритм и алгоритм A*. Разработаны программы, которые с помощью данных алгоритмов находят кратчайший путь в графе от начальной вершины до конечной.

Приложение А

Исходный код программы

Название файла: Source.cpp

```
#include <iostream>
#include <set>
#include <vector>
#include <map>
#include <list>
#include <queue>
#include <sstream>
class Grapth;
#define MAX_INT  2147483647;

class Vertex
{
public:
	char name;
	float cost;
	bool operator < (Vertex const& other) const
	{
		if (cost == other.cost)
			return name < other.name;
		return cost > other.cost;
	}

};


class Link
{
public:
	char name_2;
	float length;

	Link(char n_2, float l) : name_2(n_2), length(l) {};

	bool operator < (Link const& other) const
	{
		if (length == other.length)
			return name_2 < other.name_2;
		return length > other.length;
	}
};



void print_set(std::set<Link> const& l)
{
	for (auto i : l)
	{
		std::cout << "\t" << " name = " << i.name_2 << " length = " <<
i.length << std::endl;
	}
}
```

```
void print_graph(std::map<char, std::set<Link>> const& graph)
{

	for (auto i : graph)
	{
		std::cout << "Vertex " << i.first << " :" << std::endl;

		print_set(i.second);
	}
}

void print_graph(std::map<char, float>& graph_state)
{

	for (auto i : graph_state)
	{
		std::cout << "Vertex " << i.first << " :" << std::endl;

		std::cout << i.second << std::endl;
	}
}


float h(char x, char finish)
{
	return abs(x - finish);
}

float h_1(char x, char finish_1, char finish_2)
{
	return std::min(abs(x - finish_1), abs(x - finish_2));
}



void print_Path(char begin, char goal, std::map<char, char>& Path)
{
	if (!Path.count(goal)) {
		std::cout << "No path" << std::endl;
		return;
	}
	std::string s = "";
	s += goal;
	char current = Path[goal];
	s = current + s;
	while (current != begin)
	{
		current = Path[current];
		s = current + s;
	}
	std::cout << s << std::endl;
}


void read_graph(std::map<char, std::set<Link>>& graph)
{
	char n1;
	char n2;
```

```
	float cost;
	std::string s;
	while (getline(std::cin, s) && !s.empty())
	{

		//std::cout<<s<<std::endl;

		(std::stringstream)s >> n1 >> n2 >> cost;



		if (graph.count(n1))
		{
			graph[n1].insert(Link(n2, cost));
		}
		else
		{
			graph.insert({ n1, { Link(n2, cost) } });
		}

	}
}



std::map<char, char> DXTR(char start, char goal_1, char goal_2,
std::map<char, std::set<Link>>& graph, int& res)
{

	std::priority_queue<Vertex> queue;
	queue.push({ start, 0 });

	std::map<char, float> cost_visited;
	cost_visited.insert({ start, 0 });


	std::map<char, char> visited;
	cost_visited.insert({ start, '0' });

	while (!queue.empty())
	{
		float cur_cost = queue.top().cost;
		char cur_node = queue.top().name;
		queue.pop();

		if (cost_visited.count(goal_1) && cost_visited[goal_1] <=
cur_cost) {
			res = 1;
			break;
		}
		if (cost_visited.count(goal_2) && cost_visited[goal_2] <=
cur_cost)
		{
			res = 2;
			break;
		}
```

```
            auto next_noodes = graph[cur_node];  // Links

            for (auto i : next_noodes)
            {

                  float neigh_cost = i.length;
                  char neigh_node = i.name_2;
                  float new_cost = cost_visited[cur_node] + neigh_cost;


                  if   (!cost_visited.count(neigh_node)   ||   new_cost   <=
cost_visited[neigh_node])  // vertex not in visited or new cost < curent
cost
                  {
                        float priority = new_cost + h_1(neigh_node, goal_1,
goal_2);
                        queue.push({ neigh_node, priority });
                        //queue.push({neigh_node, new_cost}); // D


                        if (!cost_visited.count(neigh_node))
                        {
                              cost_visited.insert({              neigh_node,
new_cost });
                        }
                        else
                        {
                              cost_visited.erase(neigh_node);
                              cost_visited.insert({              neigh_node,
new_cost });
                        }
                        //cost_visited.insert_or_assign(neigh_node,
new_cost);
                        if (!visited.count(neigh_node))
                        {
                              visited.insert({ neigh_node, cur_node });
                        }
                        else
                        {

                              visited.erase(neigh_node);
                              visited.insert({ neigh_node, cur_node });
                        }
                        //visited.insert_or_assign(neigh_node, cur_node);
                  }

            }
      }
      return visited;
}


int main()
{
      std::map<char, std::set<Link>> graph;

      char start;
      char end_1;
      char end_2 = 0;
```

```
	int result = 0;

	std::cin >> start >> end_1 >> end_2;
	std::cin.ignore();
	read_graph(graph);
	//std::cout << "dd";

	//print_graph(graph);

	auto t = DXTR(start, end_1, end_2, graph, result);

	if(result == 1)
		print_Path(start, end_1, t);
	else
		print_Path(start, end_2, t);




	return 0;
}
```

Приложение Б
Тестирование

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a e e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	финиши совпадают
2.	a b e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ab	финиши не совпадают
3.	a l 1 a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3	abgenmjl	
4.	a l e a b 1	abge	

	a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3		
5.	a f f a b 1 a d 2 a f 7 c d 1 c e 4 c a 8 d f 1 e d 2 e f 5 f e 1	adf	
6.	c b f a b 1 a d 2 a f 7 c d 1 c e 4 c a 8 d f 1 e d 2	cdf	

	e f 5 f e 1		
7.	d b b a f 2 a b 5 b c 4 c d 9 d f 3 d e 7 e a 1 f e 8 f c 1	deab	
8.	a e d a f 2 a b 5 b c 4 c d 9 d f 3 d e 7 e a 1 f e 8 f c 1	afe	