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 на графах

Студент гр. 9382 _____ Савельев И.С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

На практике ознакомиться с различными алгоритмами поиска 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 посещенная вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещенной 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

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.

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
Вар. 4. Модификация A* с двумя финишами (требуется найти путь до любого из двух).

Описание алгоритма.

Жадный алгоритм:

Считывается ввод пользователя, формируется массив структур Edge, затем с помощью этого массива формируется список смежности вершин, реализованный с помощью ассоциативного контейнера map, где вершина из которой исходит ребро - ключ, а значение - вектор пар содержащих вершину и длину ребра которое входит в нее, после формирования map, вектор пар сортируется по уменьшению длин ребер. Вызывается рекурсивная функция, которая начиная со стартовой вершины рассматривает, отсортированный список смежных вершин, когда на вход функции попадет финишная вершина рекурсия завершается.

Алгоритм A*:

Считывается ввод пользователя, формируется массив структур Edge, затем с помощью этого массива формируется список смежности вершин, реализованный с помощью ассоциативного контейнера map, где вершина из которой исходит ребро - ключ, а значение - вектор пар содержащих вершину и длину ребра которое входит в нее. Также создается три контейнера map, первый хранит минимальный путь до вершины, второй информацию о том была ли посещенная вершина и третий информацию о предыдущей вершине для данной. В начале работы алгоритма в очереди с приоритетом находятся только стартовая вершина, затем в нее заносятся все вершины смежные с ней, а

стартовая отмечается как посещенная, также алгоритм работает и с другими вершинами. Если текущая стоимость пути до вершины больше то вершина обновляется и изменяется информация в соответствующих контейнерах map. Стоимость пути рассчитывается путем суммирования длин пройденных ребер до вершины и близости символов, обозначающих вершины графа, в таблице ASCII. Алгоритм прекращает работу когда закончатся вершины в очереди.

Описание структур данных представляющих граф.

Описывает ребро и две прилегающие к нему вершины.

struct Edge {

char top1; // вершина из которой выходит ребро

char top2; // вершина в которую входит ребро

double lenght; // длинна ребра

};

vector<Edge> edge_mass - массив структур Edge

map<char, vector<pair<char, double >>> nodes - ассоциативный контейнер описывающий смежные вершины. Ключ - вершина из которой исходит ребро ключ, значение - вектор пар содержащих вершину и длину ребра которое входит в нее.

Оценка сложности по памяти и времени.

Жадный алгоритм:

В худшем случае придется обойти весь граф - O|E + L|, где E - количество вершин, а L - количество ребер.

Сложность по памяти в худшем случае - O|2*(E + L)|, где E - количество вершин, а L - количество ребер.

Алгоритм A*:

В худшем случае, число вершин, исследуемых алгоритмом, растет экспоненциально по сравнению с длиной 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.

В лучшем случае, когда выбрана наиболее подходящая эвристическая функция, которая выбирает лучший путь на каждом шаге, время выполнения будет O|E + L|, где E - количество вершин, а L - количество ребер.

В худшем случае, когда эвристическая функция выбирает верное направление в последнюю очередь. Придется перебрать все возможные пути и в таком случае сложности будет - O(E^2).

В худшем случае все пути будут храниться в очереди, следовательно сложность по памяти будет экспоненциальной.

В лучшем случае будет хранится путь для вершины от начала до нее. И сложность по памяти будет O|E * (E + L)|, где E - количество вершин, а L - количество ребер.

Тестирование.

Таблица 1 результаты работы жадного алгоритма.

Входные данные	Вывод программы
a e a g 5 a b 1 g e 1	abe

b e 2	
z k z m 1 z l 5 l k 4	zlk
a b a b 10 a g 1 g m 2	ab

Таблица 2 результат работы алгоритма A*

Входные данные	Вывод программы
a e g a e 12 a g 11	ag
a c k a b 2 a l 2 b m 5 l m 1 m c 1 m k 1	almc
a e e a b 10 a c 1 b e 2	abe

Выводы.

В ходе выполнения данной лабораторной работы был реализован жадный алгоритм и алгоритм A* на языке программирования C++.

Приложение А. Исходный код программы.

Жадный алгоритм:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>
#include <map>

using namespace std;

//структура, описывающая ребро
struct Edge {
    char top1; // вершина из которой выходит ребро
    char top2; // вершина в которую входит ребро
    double lenght; // длинна ребра
};

bool Sort(const pair<char, double >& a, const pair<char, double >& b) {
    return (a.second < b.second);
}

// сортирует смежные вершины по длине ребра, от меньшего к большему
void sort_lenght(map<char, vector<pair<char, double >>>& nodes) {
     for(auto& item : nodes) {
      sort(nodes[item.first].begin(), nodes[item.first].end(), Sort);
    }
}

// формирует список смежности вершин
void   make_list(map<char,   vector<pair<char,   double   >>>&   nodes,
vector<Edge>& edge_mass) {
    char top1;
    char top2;
    double lenght;
     // edge_mass.size() - кол-во ребер
    for (int i = 0; i < edge_mass.size(); i++) {
      top1 = edge_mass[i].top1;
      top2 = edge_mass[i].top2;
      lenght = edge_mass[i].lenght;
      nodes[top1].push_back(make_pair(top2, lenght));
    }
}

// рекурсивная функция поиска пути
void   find_way(map<char,   vector<pair<char,   double   >>>&   nodes,
vector<char>& way, char top1, char top2, bool& flag) {
    // добавляем вершину в путь
    std::cout << "Добавили вершину: " << top1 << "\n\n";
    way.push_back(top1);
```

```
    // если пришли в конечную вершину
    if (top1 == top2) {
       flag = true;
       return;
    }
    std::cout << "Список смежных вершин с " << top1  << '\n';
    for (int i = 0; i < nodes[top1].size(); i++) {
       std::cout << nodes[top1][i].first << " " << nodes[top1][i].second
<< '\n';
    }
    for (int i = 0; i < nodes[top1].size(); i++) {
       find_way(nodes, way, nodes[top1][i].first, top2, flag);
       if (flag) {
            return;
      }
       // удаляем вершину из пути
       way.pop_back();
    }
}

// считывает ввод пользователя
void user_input(map<char, vector<pair<char, double >>>& nodes) {
    vector<Edge> edge_mass;
    Edge elem;
    char top1;
    char top2;
    top1 = ' ';
    double lenght;
    while (cin >> top1) {
       if (!top1 || top1 == '/') {
            break;
      }
       cin >> top2;
       cin >> lenght;
       elem.top1 = top1;
       elem.top2 = top2;
       elem.lenght = lenght;
       // добавляем в массив
       edge_mass.push_back(elem);
    }

    make_list(nodes, edge_mass);
}

int main() {
    char top1;
    char top2;
    bool flag = false;
    vector<char> way;
    map<char, vector<pair<char, double >>> nodes;
```

```
    cin >> top1;
    cin >> top2;

    user_input(nodes);
    sort_lenght(nodes);

    find_way(nodes, way, top1, top2, flag);

    for (int i = 0; i < way.size(); i++) {
       cout << way[i];
    }

    return 0;
}
```

Алгоритм A*:

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <ctime>

using namespace std;

//структура, описывающая ребро
struct Edge {
      char top1; // вершина из которой выходит ребро
      char top2; // вершина в которую входит ребро
      double lenght; // длинна ребра
};

// формирует список смежности вершин
void   make_list(map<char,   vector<pair<char,   double   >>>&   nodes,
vector<Edge>& edge_mass) {
    char top1;
    char top2;
    double lenght;
      // edge_mass.size() - кол-во ребер
    for (int i = 0; i < edge_mass.size(); i++) {
       top1 = edge_mass[i].top1;
       top2 = edge_mass[i].top2;
       lenght = edge_mass[i].lenght;
       nodes[top1].push_back(make_pair(top2, lenght));
    }
}

// суммирует все ребра в графе
int sum_edge(map<char, vector<pair<char, double >>>& nodes) {
      int sum = 0;
```

```
	for(auto& item : nodes) {
	for (int j = 0; j < nodes[item.first].size(); j++) {
		sum += nodes[item.first][j].second;
	}
	}
	return sum + 1;
}

// выводит ответ
void print_answer( map <char, char> previous_top, char top) {
	char last = top;
	std::vector<char> my_vector;
	// перебираем вершина от последней к первой
	while (previous_top[top] != '*') {
	//заносим в массив
	my_vector.push_back(previous_top[top]);
	top = previous_top[top];
	}
	// выводи вершины в правильном порядке
	for (auto it = my_vector.rbegin(); it != my_vector.rend(); ++it) {
	std::cout << *it;
	}
	// последняя вершина
	std::cout << last << '\n';

}

// переназначеам расстояние до вершин
void change_distance(map<char, vector<pair<char, double >>> nodes,
priority_queue<pair<double, char>>& top_queue, map <char, double>&
distance , map <char, char>& previous_top, char top, char top2) {
	// перебираем все вершины смежные с top
	for (int i = 0; i < nodes[top].size(); i++) {
	// если нашли путь короче предыдущего, заменяем
	if (distance[nodes[top][i].first] > distance[top] +
nodes[top][i].second) {
		distance[nodes[top][i].first] = distance[top] +
nodes[top][i].second;
		// меняем предыдущую вершину
		std::cout << "Длинна кратайшего пути до " <<
nodes[top][i].first << " изменена на " << distance[top] +
nodes[top][i].second <<'\n';
		previous_top[nodes[top][i].first] = top;
		// добовляем в очередь
		top_queue.push(make_pair(-(distance[nodes[top][i].first] +
(int)top2 - (int)nodes[top][i].first), nodes[top][i].first));
		std::cout << "Занесли в очередь " << nodes[top][i].first << "
f(x) = " << (distance[nodes[top][i].first] + (int)top2 -
(int)nodes[top][i].first) << "\n\n";
	}
	}
```

```
}

// Алгорит A*
void AStar(map<char, vector<pair<char, double >>>& nodes, char top1, char
top2) {
      priority_queue<pair<double, char>> top_queue;
      map <char, double> distance ;
      map <char, char> previous_top;
      map <char, bool> visited_top;
      int current_top;
      int min;

      double max_distance  = sum_edge(nodes);

      // заполняем map
      for(auto& item : nodes) {
      distance[item.first] = max_distance ;
      }

      for(auto& item : nodes) {
      visited_top[item.first] = false;
      }

      for(auto& item : nodes) {
      previous_top[item.first] = '!';
      }

      distance[top2] = max_distance ;
      visited_top[top2] = false;
      previous_top[top2] = '!';
      distance [top1] = 0;
      previous_top[top1] = '*';

      // добовляем стартовую вершину в очередь
      top_queue.push(make_pair(0, top1));
      std::cout << "Занесли в очередь " << top1 << " f(x) = "<< 0 <<
"\n\n";
      while (!top_queue.empty()) {
      while (1) {
            // если очередь пуста
            if (top_queue.empty()) {
                  break;
            }
            // считываем вершину из очереди
            pair<int, char> current_min = top_queue.top();
            top_queue.pop();
            current_top = current_min.second;
            min = -current_min.first;
            std::cout << "Извлекли из очереди " << (char)current_top << "
f(x) = " << min << "\n\n";
            // если вершина была посещена
```

```
            if (!visited_top[current_top]) {
                  break;
            }
      }
      if (distance[top2] < min) {
            break;
      }
      // отмечаем вершину как посещенную
      visited_top[current_top] = true;
      change_distance   (nodes,   top_queue,   distance   ,   previous_top,
current_top, top2);
      }
      // выводим ответ
      print_answer(previous_top, top2);
}

// считывает ввод пользователя
void user_input(map<char, vector<pair<char, double >>>& nodes) {
      vector<Edge> edge_mass;
    Edge elem;
    char top1;
    char top2;
    top1 = ' ';
    double lenght;
    while (cin >> top1) {
       if (!top1 || top1 == '/') {
             break;
      }
       cin >> top2;
       cin >> lenght;
       elem.top1 = top1;
       elem.top2 = top2;
       elem.lenght = lenght;
      // добавляем в массив
       edge_mass.push_back(elem);
    }
    make_list(nodes, edge_mass);
}

int main() {
      srand(static_cast<unsigned int>(time(0)));
      char top1;
      char top2;
      char top3;
      cin >> top1;
      cin >> top2;
      cin >> top3;
      map<char, vector<pair<char, double >>> nodes;
      user_input(nodes);
      // выбираev какая вершина будет финишом
      if(rand() % 2) {
```

```
        AStar(nodes, top1, top2);
        }
        else {
        AStar(nodes, top1, top3);
        }
        return 0;
}
```