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 _____ Кондратов Ю.А.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2022

Цель работы.

Написать программу, реализовывающую поиск кратчайшего пути во взвешенном графе, используя жадный алгоритм и алгоритм A*.

Задание.

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.

Вариант 8:

Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Выполнение работы.

Исходный код модулей представлен в приложении А.

1. Жадный алгоритм.

Жадный алгоритм реализован в файле greedy.cpp в функции find_min_path. Данная функция принимает на вход граф, представленный в виде списка смежности. Список смежности реализован при помощи std::map и std::vector.

В теле функции в цикле while, условием выхода из которого является равенство текущей вершины и заданной конечной вершины, на каждом шаге просматриваются все не обойдённые соседи текущей вершины. Из них

выбирается сосед, путь до которого наименьший, после чего текущая вершина помечается как обойдённая. Если у текущей вершины нет не обойдённых соседей, то возвращаемся к рассмотрению предыдущей вершины, в противном случае текущей вершиной становится выбранный сосед.

2. Алгоритм A*.

Перед запуском алгоритма, в соответствие с заданием, производится сортировка всех смежных вершин для каждой вершины графа по приоритету (функция sort_verices.

Данный алгоритм реализован в файле a_star.py в функции a_star. Функция принимает на вход граф, представленный в виде списка смежности, реализованного при помощи dict, list и tuple. Также алгоритм использует такую структуру данных как очередь с приоритетом, а точнее её стандартную реализцаия из модуля heapq.

В теле функции в цикле while, условием выхода из которого является пустота очереди (в очереди изначально находится начальная вершина, путь от которой требуется найти), сначала из очереди с приоритетом достается верхний элемент, если это конечная вершина (путь до которой требуется найти), то производится выход из цикла. Далее рассматриваются все соседи текущей вершины. Пути до всех соседей обновляются, если через текущую вершину они короче. Рассчитывается приоритет каждой вершины (эвристика в данном случае – разность между ascii кодами имён вершин). Кортеж из приоритета вершины и её имени добавляется в очередь с приоритетом.

Функция возвращает словарь, состоящий из пар {вершина: предыдущая вершина}, в данном случае под предыдущей вершиной подразумевается вершина, которая идёт перед вершиной, являющейся ключом, в найденном кратчайшем пути. По этому словарю далее восстанавливается весь путь (функция find_path).

Оценка сложности алгоритмов.

1. Жадный алгоритм.

Граф хранится в структуре map, каждая вершина извлекается ровно один раз, то есть необходимо сделать O(V) извлечений. На каждом шаге рассматриваются смежные вершины за O(E) операция. Таким образом сложность – O(V*E).

2. Алгоритм A*.

На каждом шаге алгоритма происходит извлечение из очереди с приоритетом за O(logV). В худшем случае рассматриваются все вершины за O(V). Также в худшем случае буду единожды рассмотрены все ребра графа. Таким образом сложность O(logV*V + E).

В данном случае эвристическая функция оценивает расстояние верно, только если вершины находятся в определённом порядке в графе, однако это выполнено далеко не всегда, поэтому эвристика работает весьма условно и на сложность в большинстве случае не влияет. Однако в лучшем случае эвристическая функция сократит сложность до O(V + E).

Тестирование.

Рассмотренные в тестировании случаи представлены в таблице 1.

Таблица 1 – Результаты тестирования

Входные данные	Выходные данные	Описание	Вердикт
a b a b 1.0	ab	Граничный случай для проверки случайный выходов за пределы массивов.	passed
a e a b 3.0 b c 1.0	ade	Тест из задания.	passed

c d 1.0 a d 5.0 d e 1.0			
a f a b 2.3 b c 4.523 c d 11.24 c f 30.309 d e 3.3 e f 3.4	acdef	Тест со случайным графом для проверки корректной работы с вещественными весами ребер.	passed
a z a b 2.0 b c 3.0 c z 4.0 a x 2.0 x y 3.0 y z 4.0	axyz	Тест для проверки корректной работы эвристической функции.	passed
z a b a 2.0 c b 3.0 z c 4.0 x z 2.0 y x 3.0 z y 4.0	zcba	Тест для проверки корректной работы эвристической функции.	passed
a z a b 1.0 b c 1.0 a e 4.0 e y 1.0	abcz	Проверка, что несмотря на большие значения эвристики, будет выбран кратчайший путь.	passed

c z 1.0 y z 1.0			
e a e b 1.0 d e 1.0 b d 1.0 d a 1.0 b a 4.0 a b 4.0 b c 1.0 c a 1.0	ebca	Проверка корректной обработки двунаправленных ребёр	passed

Протокол тестирования представлен на рисунке 1.

```
(venv) C:\LETI\DAA\lab2_py>pytest -v
================================= test session starts =================================
platform win32 -- Python 3.10.2, pytest-7.1.1, pluggy-1.0.0 -- C:\LETI\DAA\lab2_py\venv\
Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\LETI\DAA\lab2_py
collected 7 items

test_a_star.py::test_a_start[test_data0-ab] PASSED                                [ 14%]
test_a_star.py::test_a_start[test_data1-ade] PASSED                               [ 28%]
test_a_star.py::test_a_start[test_data2-abcdef] PASSED                            [ 42%]
test_a_star.py::test_a_start[test_data3-axyz] PASSED                              [ 57%]
test_a_star.py::test_a_start[test_data4-zcba] PASSED                              [ 71%]
test_a_star.py::test_a_start[test_data5-abcz] PASSED                              [ 85%]
test_a_star.py::test_a_start[test_data6-ebca] PASSED                              [100%]

================================= 7 passed in 0.03s ==================================
```

Рисунок 1 – Протокол тестирования

Выводы.

В результате выполнения работы были изучены алгоритмы поиска кратчайшего пути в ориентированном графе (жадный алгоритм и алгоритм A*). Была написана программа, которая считывает граф и находит в нем путь от стартовой вершины до конечной при помощи жадного алгоритма и алгоритма A*.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy.cpp

```
#include <iostream>
#include <map>
#include <vector>
#include <limits>

#define MAX_NODES_NUM 26

using namespace std;

vector<char> findMinPath(map<char, vector<pair<char, double>>> graph,
char source, char dest){
    char cur = source;
    bool visited[MAX_NODES_NUM] = {false};
    visited[cur - 'a'] = true;
    vector<char> res;
    res.push_back(source);

    while (cur != dest){
        double min = numeric_limits<double>::max();
        char next;
        bool flag = false;

        for (auto &n: graph[cur])
            if (!visited[n.first - 'a'] && n.second < min){
                min = n.second;
                next = n.first;
                flag = true;
            }

        visited[cur - 'a'] = true;

        if (!flag){
            if (!res.empty()){
                res.pop_back();
                cur = res.back();
            }
            continue;
        }

        cur = next;
        res.push_back(cur);
    }

    return res;
}

int main(){
    double weight;
    map<char, vector<pair<char, double>>> graph;

    char source, dest, e_start, e_end;
    cin >> source >> dest;
```

```
    while (cin >> e_start >> e_end >> weight) {
        graph[e_start].push_back({e_end, weight});
        if (cin.eof()) break;
    }

    vector<char> res = findMinPath(graph, source, dest);
    for (auto s: res)
        cout << s;
    return 0;
}
```

Название файла: a_start.py

```
import heapq


def heuristic(a: str, b: str) -> int:
    return abs(ord(b) - ord(a))


def sort_vertices(graph: dict[str: tuple[str, float]], goal):
    for vert in graph:
        graph[vert].sort(key=lambda x: heuristic(x[0], goal))


def find_path(goal: str, paths: dict[str: str]) -> str:
    path = goal
    prev = paths[goal]
    while prev is not None:
        path = prev + path
        prev = paths[prev]
    return path


def a_star(initial: str, goal: str, graph: dict[str: list[tuple[str,
float]]]):
    path_cost = {initial: 0}
    paths = {initial: None}
    queue = []
    heapq.heappush(queue, (0, initial))

    while len(queue):
        current = heapq.heappop(queue)[1]

        if current == goal:
            break

        for node in graph[current]:
            cost = path_cost[current] + node[1]
            if node[0] not in path_cost or cost < path_cost[node[0]]:
                path_cost[node[0]] = cost
                priority = cost + heuristic(goal, node[0])
                heapq.heappush(queue, (priority, node[0]))
                paths[node[0]] = current

    return paths
```

```
    def read() -> tuple[str, str, dict]:
        graph = {}
        initial_v, goal_v = input().split()

        while True:
            try:
                line = input()
            except EOFError:
                break
            if not line:
                break
            start, end, weight = line.split()
            weight = float(weight)
            if start in graph:
                graph[start].append((end, weight))
            else:
                graph[start] = [(end, weight)]
            if end not in graph:
                graph[end] = []

        return initial_v, goal_v, graph


    def solve(initial: str, goal: str, graph: dict[str: list[tuple[str,
float]]]):
        sort_vertices(graph, goal)
        paths = a_star(initial, goal, graph)
        return find_path(goal, paths)


    if __name__ == '__main__':
        initial, goal, graph = read()
        print(solve(initial, goal, graph))
```

Название файла: test_.py

```
    import pytest
    from a_star import solve


    @pytest.mark.parametrize("test_data, result", [
        (("a", "b", {"a": [("b", 1.0)]}), "ab"),
        (('a', 'e', {'a': [('b', 3.0), ('d', 5.0)], 'b': [('c', 1.0)],
'c': [('d', 1.0)], 'd': [('e', 1.0)],
                     'e': []}), 'ade'),
        (('a', 'f', {'a': [('b', 2.3)], 'b': [('c', 4.523)], 'c': [('d',
11.24), ('f', 30.309)], 'd': [('e', 3.3)], 'f': [],
                     'e': [('f', 3.4)]}), "abcdef"),
        (('a', 'z', {'a': [('b', 2.0), ('x', 2.0)], 'b': [('c', 3.0)],
'c': [('z', 4.0)], 'z': [], 'x': [('y', 3.0)],
                     'y': [('z', 4.0)]}), "axyz"),
        (('z', 'a', {'b': [('a', 2.0)], 'a': [], 'c': [('b', 3.0)], 'z':
[('c', 4.0), ('y', 4.0)], 'x': [('z', 2.0)],
                     'y': [('x', 3.0)]}), "zcba"),
```

```
        (('a', 'z', {'a': [('b', 1.0), ('e', 4.0)], 'b': [('c', 1.0)],
'c': [('z', 1.0)], 'e': [('y', 1.0)],
                     'y': [('z', 1.0)], 'z': []}), 'abcz'),
        (('e', 'a', {'e': [('b', 1.0)], 'b': [('d', 1.0), ('a', 4.0),
('c', 1.0)], 'd': [('e', 1.0), ('a', 1.0)],
                     'a': [('b', 4.0)], 'c': [('a', 1.0)]}), 'ebca')
    ])
    def test_a_start(test_data, result):
        assert solve(*test_data) == result
```