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*

Студентка гр. 9382 _____ Голубева В.П.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Понять, что такое жадный алгоритм, научиться его реализовывать на примере поиска пути в ориентированном графе.

Задание.

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.

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

2)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

Вариант 1. В A* вершины именуются целыми числами (в т.ч. отрицательными)

Описание алгоритма

Был реализован поиск пути в ориентированном графе жадным алгоритмом.

Граф представлялся в виде словаря списков объктов класса Edge.

Текущим решением будет список, в котором будут хранятся вершины, составляющие путь.

Главный цикл в алгоритме проходит по вершинам в графе. Берёт последнюю вершину из текущего пути и смотрит, путь до какой вершины наименьший. Записывает её в текущий путь, затем удаляет из графа. Таким образом, текущий путь будет содержать вершины, пути до которых минимальны. Если на очередном шаге не удалось найти путь из последней текущей вершины, это значит, что мы не дошли в заданную вершину, нужно «откатиться» и попробовать поискать решение, которое приводит нас к финишу другим путём.

Сложность по времени — из последней вершины мы выбираем не первую попавшуюся, как в поиске в глубину, а с минимальным весом, в худшем случае придётся обойти весь граф. Для графа с n вершинами и m ребрами — o(n*m).

Сложность по памяти — храним граф и текущий путь, если размер графа n*m, то сложность равна o(4*n*m), умножаем на 4 потому что класс Edge хранит 4 значения.

Также был реализован поиск пути в ориентированном графе с помощью алгоритма A*.

Граф представлен в виде std::map<int, std::vector<std::pair<int, float> >> , то есть значением для каждой вершины-источника является вектор, который

хранит смежные с ней вершины и расстояния до них. Ввод должен заканчиваться символом &.

Текущее решение будет храниться в переменной shortPath типа std::map, в качестве ключей будут выступать названия вершин типа int, а в качестве значений — std::pair<int, float>, где будет храниться имя родителя типа int и расстояние до вершины, которая выступает в качестве ключа.

Также у нас есть очередь с приоритетами currentQueue, которая используется для оценки минимального пути до вершины.

Вначале заносим в очередь начальную вершину, затем на каждом шаге извлекаем из очереди вершину, заносим её в std::map<int, bool> alreadyViewedVertices, и ищем её соседей, заносим их в очередь, оцениваем текущий путь до вершины и тот, который был посчитан до этого. Для каждой рассматриваемой вершины помимо расстояния до неё учитывалась также эвристика — близость вершины к конечной. Если найденный путь лучше, чем предыдущий, то обновляем соответствующее значение в shortPath.

Время выполнения поиска зависит от эвристики. В худшем случае число вершин, которые будет исследовать алгоритм будет расти экспоненциально, также можно добиться полиномиальной сложности, когда эвристика будет удовлетворять неравенству: |h(x) — h*(x)|<=O(log h*(x)), где h* - оптимальная эвристика. То есть ошибка h(x) должна расти медленнееЮ чем логарифм от оптимальной эвристики.

В лучшем случае, когда разность для любой её вершины и потомка не превышает веса ребра, а также для любой вершины эвристическая оценка меньше или равна минимальному пути до цели, то сложность будет равна O(n+m), потому что на каждом шаге мы будем прибилаться к цели.

В худшем же случае, эвристика нам не помогает приближаться к цели, поэтому придётся просмотреть все пути. В таком случае сложность будет сравнима со сложностью алгоритма Дейкстры и возрастёт до O(n^2).

В худшем случае на каждом шаге для нятой вершины придётся проверить все смежные вершины и если каждый путь в них будет короче уже посчитанного, придётся добавить их в очередь. Тогда сложность будет расти экспоненциально. По памяти будет O(b^m), где b — среднее число ветвлений.

Функции и структуры данных

Для жадного алгоритма был реализован class Edge с полями int name — имя конца ребра, int length - длина ребра, int flag — используется вершина в пути или нет, int was - посещали данную вершину или нет

def print_path(str, path)— функция, принимает строку и путь(список вершин), возвращает строку, состоящую из вершин в пути подряд

def print_dict(dic) — принимает словарь и выводит его на экран

Для A* реализован также класс Edge с полями int name - имя конца ребра, int parent — имя вершины-родителя, float heuristic — значение эвристики для данного ребра

Edge(int n, int par, float r) — конструктор класса, принимает int n — имя конца ребра, int par — имя вершины-родителя, float r — значение для эвристики

~Edge() - деструктор класса

Был реализован класс class Cmp для сравнения двух ребёр

Cmp() - конструктор

~Cmp() - деструктор

bool operator()(const Edge& a, const Edge& b) - оператор сравнения, принимает два ребра и возвращает результат их сравнения

void printPath(int current, int begin, std::map<int, std::pair<int, float>> way) - принимает вершину int current — конечная вершина для выводимого пути,

вершину int begin — начальная вершина в пути, также std::map<int, std::pair<int, float>> - граф, печатает путь между begin и current

void printQueue(std:: priority_queue <Edge, std::vector<Edge>, Cmp> currentQueue) — принимает std::priority_queue <Edge, std::vector<Edge>, Cmp> currentQueue - очередь с приоритетами b пучатает её на экран

Тестирование

Результаты тестирования первой программы можно посмотреть в приложении В.

Результаты тестирования второй программы можно посмотреть в приложении Г.

Выводы.

Было изучено что такое жадный алгоритм и алгоритм A*, написана программа, которая их реализует.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ LAB2_1

Название файла: lab2_1.py

```
class Edge:
    def __init__(self, nam, len):
        self.name = nam #name of end vertice
        self.length = len# the size of edge
        self.flag = 0# use or unuse this edge in path
        self.was = 0# has this rib been viewed

def print_path(str, path):
    for i in path:
        str+=i.name
    print(str)
def print_dict(dic):
    print("{")
    for i in dic:
        print(i)
        f = dic[i]#list

        for j in f:
            print(str(j.name)+ ' '+str(j.length)+' '+str(j.flag)+'
'+str(j.was))
    print("}")

path = input().split(' ') #list

vertice = input()
dictionary = {}
while (vertice):
    current = vertice.split(' ')
    if current[0] in dictionary:
        dictionary[current[0]].append(dict.fromkeys([current[1]],
float(current[2])))#if current source vertice alredy in dictionary
    else:
```

```
        dictionary[current[0]]=[dict.fromkeys([current[1]],
float(current[2]))]#if current source vertice not in dictionary
    try:
        vertice = input()
    except:
        break
print("Our incoming pathes")
print(dictionary)

current_path = [path[0]]#store a path of vertices
size = 1#current path size

while current_path[size-1]!=path[1]:
    print ("\nCurrent path")
    print(current_path)
    #trying to make path from the last vertice in current path
    try:
        find = dictionary[current_path[size-1]]#list of dictionaries
        print("Trying find greedy path from "+current_path[size-1])
    except:
        print("No path from "+ current_path[size-1]+", delete this
vertice from current_path")
        current_path.pop()
        size-=1
        continue

    min_size = 1000
    for i in find:# i - dict
        keys = i.keys()# get a keys for current source vertice

        for j in keys:
            if i[j] < min_size:
                min_size = i[j]
                min_key = j
                item = i
    find.remove(item)#remove vertice from dictionary
    print("Find greedy path from "+min_key)
```

```
    current_path.append(min_key)
    size+=1

str_result = "\nResult path: "
for i in current_path:
    str_result+=str(i)
print(str_result)
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ LAB2_2

Название файла: lab2_2.cpp

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

class Edge{
public:
    int name;
    int parent;
    float heuristic;
```

```
    Edge(int n, int par, float r):name(n), parent(par), heuristic(r){}
    ~Edge() = default;
};

class Cmp{
public:
    Cmp() = default;
    ~Cmp() = default;
    bool operator()(const Edge& a, const Edge& b);
};

bool Cmp::operator()(const Edge& a, const Edge& b){// a compare
function to priority queue
    if (a.heuristic == b.heuristic){
        return a.name < b.name;
    }
    return a.heuristic > b.heuristic;
}

void printPath(int current, int begin, std::map<int, std::pair<int,
float>> way){
    if (begin == current){
        std::cout << current<< ' ';
        return ;
    }
    printPath(way[current].first, begin, way);
    std::cout << current<< ' ';
}

void printQueue(std:: priority_queue <Edge, std::vector<Edge>, Cmp>
currentQueue){
    std::cout<<"Current queue: ";
    while (!currentQueue.empty()){
        auto vertice = currentQueue.top();
        std::cout<<vertice.parent<<':'<<" ("<< vertice.name<<", "<<
vertice.heuristic <<"), ";
```

```
        currentQueue.pop();
    }
    std::cout<<"\n";
}

int main(){

    std::map<int, std::vector<std::pair<int, float> >> pathesGraph;
    std::vector <Edge> Edges;// a vector to save a pop vertice
    std:: priority_queue <Edge, std::vector<Edge>, Cmp> currentQueue;
    std::map<int, std::pair<int, float>> shortPath; // a shortes paths
to vertices
    std::map<int, bool> alreadyViewedVertices;

    int begin, result;//the path to find

    std::cin >> begin >> result;

    int start, finish;
    float length = 0.0;

    while (std::cin >> start){
        if (start == '&')
            break;
        std::cin >> finish >> length;
        pathesGraph[start].push_back(std::make_pair(finish, length));
    }

    currentQueue.push(Edge(begin, '\0', float(result - begin)));

    while(!currentQueue.empty()){
        printQueue(currentQueue);
        if (currentQueue.top().name == result){ // if find a result
vertice
            std::cout<<"Find path! Result path: ";
            printPath(result, begin, shortPath);
            std::cout<<"\n";
```

```
            return 0;
        }

        for (int i = 0 ; i < 1 && !currentQueue.empty(); i++){
            Edge current = currentQueue.top();
            std::cout<<"Consider a vertice: "<<current.parent<<':'<<",
("<< current.name<<", "<< current.heuristic <<") \n";

            if (current.name == result){
                continue;
            }

            Edges.push_back(current);
            currentQueue.pop();
        }

        int size = Edges.size();

        for(int i = 0; i < size; i++){// consider a taken off vertices

            Edge currentEdge = Edges[i];
            alreadyViewedVertices[currentEdge.name] = true;
            std::cout<<"Consider a adjacent vertices for virtice
'"<<currentEdge.name<<"'\n";
            for (int j = 0; j < pathesGraph[currentEdge.name].size();
j++){// consider all adjacent vertices

                std::pair<int, float> newEdge =
pathesGraph[currentEdge.name][j];
                std::cout<<"Check ("<< newEdge.first<<",
"<<newEdge.second<<")\n";
                if (alreadyViewedVertices[newEdge.first]){
                    continue;
                }
                float newPath = newEdge.second +
shortPath[currentEdge.name].second;
```

```
                if (shortPath[newEdge.first].second == 0 || newPath <
shortPath[newEdge.first].second){//check path from this vertice
                    std::cout<<"Find the shorter path or new path to
"<<newEdge.first<<" from "<< currentEdge.name<<" with length "<<
newPath <<" \n";
                    //added if he path more short or vertice not
considered earlier
                    shortPath[newEdge.first].first = currentEdge.name;
                    shortPath[newEdge.first].second = newPath;
                    currentQueue.push(Edge(newEdge.first,
currentEdge.name, shortPath[newEdge.first].second + float(result-
newEdge.first)));
                }
            }
        }
        std::cout<<"\n";
        Edges.clear();
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ В

ТЕСТИРОВАНИЕ ЖАДНОГО АЛГОРИТМА

Входные данные	Выходные данные
a z a b 1 a c 3 b y 6 c y 1 y z 1	Our incoming pathes {'a': [{'b': 1.0}, {'c': 3.0}], 'b': [{'y': 6.0}], 'c': [{'y': 1.0}], 'y': [{'z': 1.0}]} Current path ['a'] Trying find greedy path from a

	Find greedy path from b Current path ['a', 'b'] Trying find greedy path from b Find greedy path from y Current path ['a', 'b', 'y'] Trying find greedy path from y Find greedy path from z Result path: abyz
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Our incoming pathes {'a': [{'b': 3.0}, {'d': 5.0}], 'b': [{'c': 1.0}], 'c': [{'d': 1.0}], 'd': [{'e': 1.0}]} Current path ['a'] Trying find greedy path from a Find greedy path from b Current path ['a', 'b'] Trying find greedy path from b Find greedy path from c Current path ['a', 'b', 'c'] Trying find greedy path from c Find greedy path from d Current path

	['a', 'b', 'c', 'd'] Trying find greedy path from d Find greedy path from e Result path: abcde
a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	Our incoming pathes {'a': [{'b': 1.0}, {'d': 9.0}, {'e': 1.0}], 'b': [{'c': 9.0}], 'c': [{'d': 3.0}], 'e': [{'d': 3.0}]} Current path ['a'] Trying find greedy path from a Find greedy path from b Current path ['a', 'b'] Trying find greedy path from b Find greedy path from c Current path ['a', 'b', 'c'] Trying find greedy path from c Find greedy path from d Result path: abcd
a b a b 1.0 a c 1.0	Our incoming pathes {'a': [{'b': 1.0}, {'c': 1.0}]} Current path ['a'] Trying find greedy path from a Find greedy path from b

	Result path: ab
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	Our incoming pathes {'a': [{'b': 3.0}, {'c': 1.0}, {'g': 8.0}], 'b': [{'d': 2.0}, {'e': 3.0}], 'd': [{'e': 4.0}], 'e': [{'a': 3.0}, {'f': 2.0}], 'f': [{'g': 1.0}]} Current path ['a'] Trying find greedy path from a Find greedy path from c Current path ['a', 'c'] No path from c, delete this vertice from current_path Current path ['a'] Trying find greedy path from a Find greedy path from b Current path ['a', 'b'] Trying find greedy path from b Find greedy path from d Current path ['a', 'b', 'd'] Trying find greedy path from d Find greedy path from e Current path ['a', 'b', 'd', 'e'] Trying find greedy path from e

	Find greedy path from f Current path ['a', 'b', 'd', 'e', 'f'] Trying find greedy path from f Find greedy path from g Result path: abdefg

ПРИЛОЖЕНИЕ Г

ТЕСТИРОВАНИЕ АЛГОРИТМА A*

Входные данные	Выходные данные
10 14 10 11 3.0 11 12 1.0 12 13 1.0 10 13 5.0 13 14 1.0	Current queue: 0: (10, 4), Consider a vertice: 0:, (10, 4) Consider a adjacent vertices for virtice '10' Check (11, 3) Find the shorter path to 11 from 10 with length 3 Check (13, 5) Find the shorter path to 13 from 10 with length 5

	Current queue: 10: (13, 6), 10: (11, 6), Consider a vertice: 10:, (13, 6) Consider a adjacent vertices for virtice '13' Check (14, 1) Find the shorter path to 14 from 13 with length 6 Current queue: 13: (14, 6), 10: (11, 6), Find path! Result path: 10 13 14
10 11 10 11 1.0 10 12 1.0	Current queue: 0: (10, 1), Consider a vertice: 0:, (10, 1) Consider a adjacent vertices for virtice '10' Check (11, 1) Find the shorter path to 11 from 10 with length 1 Check (12, 1) Find the shorter path to 12 from 10 with length 1 Current queue: 10: (12, 0), 10: (11, 1), Consider a vertice: 10:, (12, 0) Consider a adjacent vertices for virtice '12' Current queue: 10: (11, 1), Find path! Result path: 10 11
1 22 1 11 1.0 1 22 50.0 1 -5 2.0 -5 22 10.0	Current queue: 0: (1, 21), Consider a vertice: 0:, (1, 21) Consider a adjacent vertices for virtice '1' Check (11, 1) Find the shorter path to 11 from 1 with length 1 Check (22, 50) Find the shorter path to 22 from 1 with length 50 Check (-5, 2) Find the shorter path to -5 from 1 with length 2

	Current queue: 1: (11, 12), 1: (-5, 29), 1: (22, 50), Consider a vertice: 1:, (11, 12) Consider a adjacent vertices for virtice '11' Current queue: 1: (-5, 29), 1: (22, 50), Consider a vertice: 1:, (-5, 29) Consider a adjacent vertices for virtice '-5' Check (22, 10) Find the shorter path to 22 from -5 with length 12 Current queue: -5: (22, 12), 1: (22, 50), Find path! Result path: 1 -5 22
-1 -5 -1 33 1.0 -1 -6 3.0 -6 -5 2.0 -5 22 10.0	Current queue: 0: (-1, -4), Consider a vertice: 0:, (-1, -4) Consider a adjacent vertices for virtice '-1' Check (33, 1) Find the shorter path to 33 from -1 with length 1 Check (-6, 3) Find the shorter path to -6 from -1 with length 3 Current queue: -1: (33, -37), -1: (-6, 4), Consider a vertice: -1:, (33, -37) Consider a adjacent vertices for virtice '33' Current queue: -1: (-6, 4), Consider a vertice: -1:, (-6, 4) Consider a adjacent vertices for virtice '-6' Check (-5, 2) Find the shorter path to -5 from -6 with length 5 Current queue: -6: (-5, 5), Find path! Result path: -1 -6 -5
a l	Current queue: 0: (10, 11),

a b 1	
a f 3	Consider a vertice: 0:, (10, 11)
b c 5	Consider a adjacent vertices for virtice '10'
b g 3	Check (11, 1)
f g 4	Find the shorter path to 11 from 10 with length 1
c d 6	Check (15, 3)
d m 1	Find the shorter path to 15 from 10 with length 3
g e 4	
e h 1	Current queue: 10: (15, 9), 10: (11, 11),
e n 1	Consider a vertice: 10:, (15, 9)
n m 2	Consider a adjacent vertices for virtice '15'
g i 5	Check (16, 4)
i j 6	Find the shorter path to 16 from 15 with length 7
i k 1	
j l 5	Current queue: 10: (11, 11), 15: (16, 12),
m j 3	Consider a vertice: 10:, (11, 11)
&	Consider a adjacent vertices for virtice '11'
	Check (12, 5)
	Find the shorter path to 12 from 11 with length 6
	Check (16, 3)
	Find the shorter path to 16 from 11 with length 4
	Current queue: 11: (16, 9), 15: (16, 12), 11: (12, 15),
	Consider a vertice: 11:, (16, 9)
	Consider a adjacent vertices for virtice '16'
	Check (14, 4)
	Find the shorter path to 14 from 16 with length 8
	Check (18, 5)
	Find the shorter path to 18 from 16 with length 9
	Current queue: 16: (18, 12), 15: (16, 12), 16: (14, 15), 11: (12, 15),
	Consider a vertice: 16:, (18, 12)
	Consider a adjacent vertices for virtice '18'

	Check (19, 6) Find the shorter path to 19 from 18 with length 15 Check (20, 1) Find the shorter path to 20 from 18 with length 10 Current queue: 18: (20, 11), 15: (16, 12), 16: (14, 15), 11: (12, 15), 18: (19, 17), Consider a vertice: 18:, (20, 11) Consider a adjacent vertices for virtice '20' Current queue: 15: (16, 12), 16: (14, 15), 11: (12, 15), 18: (19, 17), Consider a vertice: 15:, (16, 12) Consider a adjacent vertices for virtice '16' Check (14, 4) Check (18, 5) Current queue: 16: (14, 15), 11: (12, 15), 18: (19, 17), Consider a vertice: 16:, (14, 15) Consider a adjacent vertices for virtice '14' Check (17, 1) Find the shorter path to 17 from 14 with length 9 Check (23, 1) Find the shorter path to 23 from 14 with length 9 Current queue: 14: (23, 7), 14: (17, 13), 11: (12, 15), 18: (19, 17), Consider a vertice: 14:, (23, 7) Consider a adjacent vertices for virtice '23' Check (22, 2) Find the shorter path to 22 from 23 with length 11 Current queue: 23: (22, 10), 14: (17, 13), 11: (12, 15), 18: (19, 17), Consider a vertice: 23:, (22, 10) Consider a adjacent vertices for virtice '22'

	Check (19, 3) Find the shorter path to 19 from 22 with length 14 Current queue: 14: (17, 13), 11: (12, 15), 22: (19, 16), 18: (19, 17), Consider a vertice: 14:, (17, 13) Consider a adjacent vertices for virtice '17' Current queue: 11: (12, 15), 22: (19, 16), 18: (19, 17), Consider a vertice: 11:, (12, 15) Consider a adjacent vertices for virtice '12' Check (13, 6) Find the shorter path to 13 from 12 with length 12 Current queue: 22: (19, 16), 18: (19, 17), 12: (13, 20), Consider a vertice: 22:, (19, 16) Consider a adjacent vertices for virtice '19' Check (21, 5) Find the shorter path to 21 from 19 with length 19 Current queue: 18: (19, 17), 19: (21, 19), 12: (13, 20), Consider a vertice: 18:, (19, 17) Consider a adjacent vertices for virtice '19' Check (21, 5) Current queue: 19: (21, 19), 12: (13, 20), Find path! Result path: 10 11 16 14 23 22 19 21