**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. 1304 |  | Байков Е.С. |
| Преподаватель |  | Шевелева А.М. |

Санкт-Петербург

2023

## Цель работы.

Изучение жадного алгоритма и алгоритма А\*.

## Задание.

Для жадного алгоритма:

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 дешевым из последней посещенной вершины. Каждая вершина в графе имеет буквенное обозначение («a», «b», «c»…), каждое ребро имеет неотрицательный вес.

Для алгоритма A\*:

Разработайте программу, которая решает задачу построения пути в ориентированном графе методом А\*. 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.

Входные и выходные данные:

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.

## Основные теоретические положения.

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Алгоритм A\* — в информатике и математике, алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть, как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

## Выполнение работы.

В ходе выполнения лабораторной работы был создан класс *Graph*, в котором и осуществляется основная реализация алгоритмов. Были описаны методы, выполняющие требуемые в задании алгоритмы, – жадный алгоритм и A\*.

**Структура**

Далее описаны классы, методы и функции используемые в коде программы для решения поставленной задачи.

1. Класс *Graph*: данный класс хранит в себе стартовый и конечный узел пути, а также информацию о дугах графа в виде словаря, где ключом является начальный узел дуги, а в качестве значения список списков из двух элементов. Первый элемент внутреннего списка – конечный узел, а второй элемент – вес дуги. Внутри класса реализованы методы: *get\_neighbours*, *heuristic\_function*, *a\_star\_algorithm*, *greedy\_algorithm* и *\_\_init\_\_*.
2. Метод *get\_neighbours*: принимает на вход узел графа и возвращает список из смежных ему узлов и весов соответствующих ребер. Если же переданный узел является листом, то возвращаемым значением будет None.
3. Метод *heuristic\_function*: эвристическая функция, которая возвращает разницу между значениями ASCII кодов переданного в нее и конечного узлов. Метод необходим для реализации алгоритма A\*.
4. Метод *a\_star\_algorithm*: реализует алгоритм поиска кратчайшего пути A\*. Внутри метода созданы открытое и закрытое множества (*open\_set*, *closed\_set*), в первом находится стартовый узел. Затем были созданы два словаря *distance* – хранит в себе пары узел (ключ) и длину пути до стартовой вершины (значение), и *adjacent\_nodes*, хранящий в себе узел назначения в качестве ключа и стартовый узел (откуда осуществлен переход в узел назначения) в качестве значения. После начинается цикл *while*, который длится до тех пор, пока в открытом множестве есть узлы требующие проверки. Внутри цикла происходит поиск минимального (по сумме эвристики и расстояния) узла из открытого множества. Если вдруг такого узла не существует, то пути не существует и метод возвращает *None*. Затем идет проверка на то, не является ли выбранный узел финишным узлом. При достижении финиша происходит проход по ключам и значениям списка смежных узлов, по которым строится путь от финиша к старту, а затем возвращается инвертированный путь, т.е. от старта к финишу. В ином случае происходит проверка списка смежных узлов с помощью метода *get\_neighbours*. При отсутствии смежных узлов из открытого множества узел удаляется и добавляется в закрытый, чтобы его больше не просматривать, затем осуществляется переход к следующей итерации. При наличии происходит просмотр всех узлов (*node*) и весов (*weight*), которые хранятся в списке *neighbours*. Если *node* не находится в открытом и закрытом списках, значит данный узел не просмотрен и должен быть добавлен в открытое множество, а также обновлены значения в словарях *adjacent\_nodes* и *distance*. Если значение *distance* от узла в открытом или закрытом множестве больше, чем значение расстояния текущего узла и добавляемого веса, то значения обновляются, т.е. расстояние *node* становится равным значению расстояния текущего узла и добавляемого веса, а также происходит перенаправление пути из *current\_node* в *node* (*adjacent\_nodes[node] = current\_node*) если же *node* находится в закрытом списке, то из него происходит удаление этого узла и добавление его в открытое множество. После цикла текущий узел удаляется из открытого множества и добавляется в закрытое. Если верхний цикл завершился, а путь не был найден, то возвращается *None*.
5. Метод *greedy\_algorithm*: для каждого ключа из набора дуг списки-значения сортируются по весу дуги. Заводится переменная *path* равная стартовому узлу. До тех пор, пока конечная вершина в пути не будет равна финишной, будет добавляться первый узел из набора смежных узлов, при условии того что он уже не находится в пути. Если данный узел уже лежит в пути, то он извлекается из набора всех дуг, для того чтобы удалить цикличность. Метод возвращает значение переменной *path*.
6. Метод *\_\_init\_\_*: конструктор класса, принимает набор ребер в виде словаря, стартовый и финишный узлы, а также инициализирует данные поля класса.
7. Функция *solve*: функция считывает стартовую и конечную позицию, а также заполняется словарь с ребрами *arcs\_kit*. Был создан объект типа *Graph*, в который передаются словарь ребер, стартовый узел и финишный узел. Печатает результат методов *greedy\_algorithm* и *a\_star\_algorithm*.
8. Функция *main*: вызывает функцию *solve*.

Разработанный программный код смотреть в приложении A.

## Выводы.

Была разработана программа, реализующая алгоритмы – жадный и A\*, для поиска кратчайшего пути в графе. Успешно пройдены тесты на платформе *Stepik*. Оба алгоритма представлены в виде методов класса *Graph*. Жадный алгоритм реализован итеративно и был немного оптимизирован путем сортировки набора смежных узлов. В алгоритме A\* использована эвристическая функция, которая была приведена на платформе *Stepik* – разность символов, текущего и конечного узлов.

# Приложение А Исходный код программы

Название файла: main.py

# Graph class stores in itself start and end nodes and kit of arcs

class Graph:

# Graph constructor that creates an object of the Graph type

def \_\_init\_\_(self, arcs\_kit, start\_node, finish\_node) -> None:

self.start\_node = start\_node

self.arcs\_kit = arcs\_kit

self.finish\_node = finish\_node

# method returns list which includes adjacent nodes and weight of arc

def get\_neighbours(self, node):

return self.arcs\_kit[node] if node in self.arcs\_kit else None

# method that returns heuristic approximation

def heuristic\_function(self, node):

return ord(self.finish\_node) - ord(node)

# method implements A\* algorithm

def a\_star\_algorithm(self):

open\_set = set(self.start\_node) # stores nodes requiring consideration

closed\_set = set() # stores checked nodes

distance = {} # stores node as key and distance to the start as value

distance[self.start\_node] = 0

adjacent\_nodes = {} # stores destination node as key and start node as value

adjacent\_nodes[self.start\_node] = self.start\_node

while len(open\_set) > 0:

current\_node = None

for node in open\_set:

if current\_node == None or\

distance[node] + self.heuristic\_function(node) <\

distance[current\_node] + self.heuristic\_function(current\_node):

current\_node = node

if not current\_node:

return None

if current\_node == self.finish\_node:

reverse\_path = ''

while adjacent\_nodes[current\_node] != current\_node:

reverse\_path += current\_node

current\_node = adjacent\_nodes[current\_node]

reverse\_path += current\_node

return reverse\_path[::-1]

neighbours = self.get\_neighbours(current\_node)

if neighbours == None:

open\_set.remove(current\_node)

closed\_set.add(current\_node)

continue

for (node, weight) in neighbours:

if node not in open\_set and node not in closed\_set:

open\_set.add(node)

adjacent\_nodes[node] = current\_node

distance[node] = distance[current\_node] + weight

elif distance[node] > distance[current\_node] + weight:

distance[node] = distance[current\_node] + weight

adjacent\_nodes[node] = current\_node

if node in closed\_set:

closed\_set.remove(node)

open\_set.add(node)

open\_set.remove(current\_node)

closed\_set.add(current\_node)

return None

# method implements greedy algorithm

def greedy\_algorithm(self):

for node in self.arcs\_kit:

self.arcs\_kit[node].sort(key=lambda x: x[1])

path = self.start\_node

while path[-1] != self.finish\_node:

current\_last\_node = path[-1]

if self.arcs\_kit[current\_last\_node][0][0] not in self.arcs\_kit \

and self.arcs\_kit[current\_last\_node][0][0] != self.finish\_node:

self.arcs\_kit[current\_last\_node].pop(0)

path += self.arcs\_kit[current\_last\_node][0][0]

return path

# function that reads input data and creates the Graph object

# then it outputs the returned values of the algorithms

def solve():

start\_node, finish\_node = input().split()

arcs\_kit = {}

while True:

try:

arc = input().split()

if arc[0] in arcs\_kit:

arcs\_kit[arc[0]].append([arc[1], float(arc[2])])

else:

arcs\_kit[arc[0]] = [[arc[1], float(arc[2])]]

except:

break

graph = Graph(arcs\_kit, start\_node, finish\_node)

print(graph.greedy\_algorithm())

print(graph.a\_star\_algorithm())

def main():

solve()

main()