**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 А\*

|  |  |  |
| --- | --- | --- |
| Студент гр. 1304 |  | Андреев В.В |
| Преподаватель |  | Шевелева А.М. |

Санкт-Петербург

2023

## Цель работы.

Решить поставленные задачи по поиску пути в графе при помощи жадного алгоритма и алгоритма А\*.

## 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.

Разработайте программу, которая решает задачу построения кратчайшего 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.

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.

## Выполнение работы.

Используемые классы и функции:

Класс *MapNode* – хранит информацию об узле графа, имена вершин, в которые можно попасть из текущей и веса соответствующих дуг.

Метод *MapNode::Sort –* сортирует дуги по весам и формирует закешированный список имен соответствующих вершин.

Метод *MapNode::AddPath –* добавляет связь с новой вершиной.

Класс *MapGraph* – хранит граф и контекстную информацию для поиска пути.

Метод *MapGraph::AddPath* – добавляет связь между узлами графа.

Метод *MapGraph::Sort* – сортирует дуги для всех вершин.

Метод *MapGraph::FindPathGreedy –* запуск жадного алгоритма поиска пути.

Метод *MapGraph::FindPathAStar -* запуск алгоритма А\* для поиска пути.

Принцип работы жадного алгоритма:

1. Всегда выбирается вершина с наименьшим весом и рекурсивно повторяется поиск пока не найдется конечная или не останется вершин вне списка посещенных.

Принцип работы алгоритма А\*:

1. Заводится список рассматриваемых вершин.
2. Поиск продолжается пока список не пуст или не нашли конечную вершину.
3. Выбор следующей вершины осуществляется путем имперических оценок. Сумма стоимости пути до текущей вершины + стоимость рассматриваемой дуги + эвристическая оценка до конечной вершины (близость символов в таблицы ASCII).
4. При рассмотрении вершины сохраняется информация об имени вершины, из которой попали в текущую.
5. Путь восстанавливается при помощи обратного прохода по именам вершин, от конечной до начальной. Ответная строка инвертируется.

Исходный код приведен в приложении А.

## Выводы.

Реализованы 2 алгоритма по поиску пути в ориентированном графе: жадный алгоритм и алгоритм А\*. Успешно пройдены тесты на платформе Stepik для обоих алгоритмов. Реализация требуемых алгоритмов заключена в одном классе. Жадный алгоритм реализован рекурсивно. В алгоритме А\* использована эвристика, приведенная в условии задания на Stepik, а именно — модуль разности символов, характеризующих идентификатор узла графа, в таблице ASCII.

# Приложение А Исходный код программы

Название файла: main.py

import functools

INF\_WEIGHT = 999999

'''

Element of graph.

'''

class MapNode:

def \_\_init\_\_(self) -> None:

self.\_\_PathsDict = dict[str, float]()

self.\_\_SortedNames = list[str]()

'''

Add link to other graph node.

@param NodeName - Name of node to link.

@param Weight - Weight of link between current node and target.

'''

def AddPath(self, NodeName: str, Weight: float) -> None:

self.\_\_PathsDict[NodeName] = Weight

'''

Sort links by its weight.

'''

def Sort(self) -> None:

self.\_\_SortedNames = list(map(lambda x: x[0], sorted(self.\_\_PathsDict.items(), key = lambda x: x[1])))

'''

@return list of sorted nodes names by links weight.

'''

def GetSortedNames(self) -> list[str]:

return self.\_\_SortedNames

'''

@return weight of link between current node and target.

'''

def GetWeightFor(self, NodeName: str) -> float:

if not NodeName in self.\_\_PathsDict:

return INF\_WEIGHT

return self.\_\_PathsDict[NodeName]

'''

Graph of nodes.

'''

class MapGraph:

def \_\_init\_\_(self) -> None:

self.\_\_PathDict = dict[str, MapNode]()

self.\_\_VisitedNodes = list[str]()

'''

Add link between nodes.

@param NodeNameA - Name of first node.

@param NodeNameB - Name of second nnode.

@param Weight - Weight of link between first and second nodes.

'''

def AddPath(self, NodeNameA: str, NodeNameB: str, Weight: float) -> None:

if not NodeNameA in self.\_\_PathDict:

self.\_\_PathDict[NodeNameA] = MapNode()

self.\_\_PathDict[NodeNameA].AddPath(NodeNameB, Weight)

'''

Sort links by weight of all nodes in graph.

'''

def Sort(self):

for LNode in self.\_\_PathDict.keys():

self.\_\_PathDict[LNode].Sort()

'''

@return dict of graph nodes.

'''

def GetPathDict(self) -> dict[str, MapNode]:

return self.\_\_PathDict

'''

Greedy algorithm to find path between Start and End nodes.

@param StartName - Name of start node.

@param EndName - Name of end node.

@return path as str.

'''

def FindPathGreedy\_Process(self, StartName: str, EndName: str) -> str:

LResult = StartName

if StartName == EndName:

return LResult

for S in self.\_\_PathDict[StartName].GetSortedNames():

if S == EndName:

return LResult + S

if (S in self.\_\_VisitedNodes) or (not S in self.\_\_PathDict):

continue

self.\_\_VisitedNodes.append(S)

LResult += self.FindPathGreedy\_Process(S, EndName)

break

return LResult

'''

A\* algorithm to find path between Start and End nodes.

@param StartName - Name of start node.

@param EndName - Name of end node.

@return path as str.

'''

def FindPathAStar\_Process(self, StartName: str, EndName: str) -> str:

LPotentialNodes = [(StartName, 0.0, 0.0)]

LNodesPath = {StartName : ""} # to - from

while len(LPotentialNodes) != 0:

LNodeName, LNodeWeight, LNodeEvristicWeight = LPotentialNodes[-1]

LPotentialNodes.pop()

self.\_\_VisitedNodes.append(LNodeName)

if LNodeName == EndName:

break

if not LNodeName in self.\_\_PathDict:

continue

for LPotentialNodeName in self.\_\_PathDict[LNodeName].GetSortedNames():

if LPotentialNodeName in self.\_\_VisitedNodes:

continue

LPotentialNodeWeight = LNodeWeight + self.\_\_PathDict[LNodeName].GetWeightFor(LPotentialNodeName)

LPotentialNodeEvristicWeight = ord(EndName) - ord(LPotentialNodeName)

tmp = [x for x in LPotentialNodes if x[0] == LPotentialNodeName]

if len(tmp) != 0:

if tmp[0][1] + tmp[0][2] > LPotentialNodeWeight + LPotentialNodeEvristicWeight:

LPotentialNodes.remove(tmp[0])

LNodesPath[LPotentialNodeName] = LNodeName

LPotentialNodes.append((LPotentialNodeName, LPotentialNodeWeight, LPotentialNodeEvristicWeight))

else:

LNodesPath[LPotentialNodeName] = LNodeName

LPotentialNodes.append((LPotentialNodeName, LPotentialNodeWeight, LPotentialNodeEvristicWeight))

def Compare(x, y) -> int:

if x[1] + x[2] == y[1] + y[2]:

return ord(x[0]) - ord(y[0])

return (x[1] + x[2]) - (y[1] + y[2])

LPotentialNodes.sort(key = functools.cmp\_to\_key(Compare), reverse = True)

if not EndName in LNodesPath:

return ""

LNodeName = EndName

LResult = LNodeName

while LNodeName != StartName:

LNodeName = LNodesPath[LNodeName]

LResult += LNodeName

return LResult[::-1]

'''

Main method to start greedy algorithm.

@param StartName - Name of start node.

@param EndName - Name of end node.

@return path as str.

'''

def FindPathGreedy(self, StartName: str, EndName: str) -> str:

self.Sort()

self.\_\_VisitedNodes.clear()

return self.FindPathGreedy\_Process(StartName, EndName)

'''

Main method to start A\* algorithm.

@param StartName - Name of start node.

@param EndName - Name of end node.

@return path as str.

'''

def FinPathAStar(self, StartName: str, EndName: str) -> str:

self.Sort()

self.\_\_VisitedNodes.clear()

return self.FindPathAStar\_Process(StartName, EndName)

if \_\_name\_\_ == "\_\_main\_\_":

Map = MapGraph()

StartName, EndName = input().split()

while True:

try:

N1, N2, W = input().split()

Map.AddPath(N1, N2, float(W))

except:

break

print(Map.FinPathAStar(StartName, EndName))