**Name: Mohammad Zain Abbas**

**Reg #: 6865**

**DE-36 (CE), Syndicate: A**

**LAB 7 JOURNEL**

**Equipment Used:** Notebook Computer, Python IDLE 3.6

**Lab Tasks:**

1 Study ‘queue’ and use it to develop priority queues in python and check if it’s working properly.

**SOLUTION CODE:**

# -\*- coding: utf-8 -\*-

"""

Created on Mon Nov 6 09:12:51 2017

@author: umerm

"""

import queue as PQ

q = PQ.PriorityQueue()

q.put(100)

q.put(10)

q.put(50)

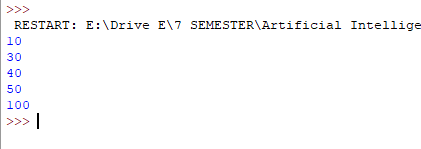
q.put(40)

q.put(30)

while not q.empty():

print(q.get());

**OUTPUT:**



2.Implement A\* algorithm in python for following graph:

Graph1: Start Node: Arad, Goal: Bucharest

**SOLUTION CODE:**

# -\*- coding: utf-8 -\*-

"""

Created on Mon Nov 13 09:15:47 2017

@author: umerm

"""

from queue import PriorityQueue

heuristics={'Arad':366,

'Bucharest':0,

'Craiova':160,

'Dobreta':242,

'Eforie':161,

'Fagaras':178,

'Giurgiu':77,

'Hirsova':151,

'Lasi':226,

'Lugoj':244,

'Mehadia':241,

'Neamt':234,

'Oradea':380,

'Pitesti':98,

'Rimnicu Vilcea':193,

'sibiu':253,

'Timisoara':329,

'Urziceni':80,

'Vaslui':199,

'Zerind':374,

};

def A\_Star\_Search(Graph,Start\_Node,Goal):

PQ=PriorityQueue();

PQ.put((0,Start\_Node));

came\_From={};

cost\_so\_Far={};

came\_From[Start\_Node]=None;

cost\_so\_Far[Start\_Node]=0;

while not PQ.empty():

current\_Node=PQ.get();

for neighbours in Graph[current\_Node[1]]:

for key in neighbours.keys():

new\_Cost=cost\_so\_Far[current\_Node[1]]+neighbours[key];

if key not in cost\_so\_Far or new\_Cost < cost\_so\_Far[key]:

cost\_so\_Far[key]=new\_Cost;

priority=new\_Cost+heuristics[key];

PQ.put((priority,key));

came\_From[key]=current\_Node[1];

return re\_ContructPath(came\_From,Start\_Node,Goal);

def re\_ContructPath(came\_From,start,goal):

current=goal;

path=[];

while current is not start:

path = path + [current];

current=came\_From[current];

path = path + [start];

path\_reverse=path[::-1];

return path\_reverse

def main():

Graph={'Arad':[{'Zerind':75},{'Timisoara':118},{'sibiu':140}],

'Zerind':[{'Oradea':71},{'Arad':75}],

'Timisoara':[{'Lugoj':111},{'Arad':118}],

'sibiu':[{'Arad':140},{'Oradea':151},{'Fagaras':99},{'Rimnicu Vilcea':80}],

'Oradea':[{'Zerind':71},{'sibiu':151}],

'Lugoj':[{'Mehadia':70},{'Timisoara':111}],

'Mehadia':[{'Lugoj':70},{'Dobreta':75}],

'Dobreta':[{'Mehadia':75},{'Craiova':120}],

'Rimnicu Vilcea':[{'Craiova':146},{'Pitesti':97},{'sibiu':80}],

'Craiova':[{'Dobreta':120},{'Rimnicu Vilcea':146},{'Pitesti':138}],

'Fagaras':[{'Bucharest':211},{'sibiu':99}],

'Pitesti':[{'Craiova':138},{'Rimnicu Vilcea':97},{'Bucharest':101}],

'Bucharest':[{'Fagaras':211},{'Pitesti':101},{'Giurgiu':90},{'Urziceni':85}],

'Giurgiu':[{'Bucharest':90}],

'Urziceni':[{'Bucharest':85},{'Hirsova':98},{'Vaslui':142}],

'Eforie':[{'Hirsova':86}],

'Hirsova':[{'Urziceni':98},{'Eforie':86}],

'Vaslui':[{'Urziceni':142},{'Lasi':92}],

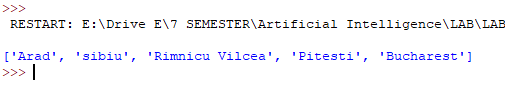
'Lasi':[{'Vaslui':92},{'Neamt':87}],

'Neamt':[{'Lasi':87}]};

print(A\_Star\_Search(Graph,'Arad','Bucharest'));

main();

**OUTPUT:**



**3. Write a script to decompose the given imaginto an undirected graph where the pixel**

**represents the vertices and adjacent vertices are connected to each other via 4-connectivity**

**and the cost on edges between adjacent nodes is their absolute intensity differences. The**

**heuristic values should be calculated by taking the Manhattan distce for each pixel**

**coordinates. The Manhattan distance between two points (x 0 ,y 0 ) and (x 1 ,y 1 ) can be**

**calculated as:**

**D = |y 1 − y 0 | + |x 1 − x 0 |**

**Use A\* algorithm to traverse decomposed image starting from pixel 150 to pixel 165.**

|  |  |  |
| --- | --- | --- |
| 150 | 2 | 5 |
| 80 | 145 | 45 |
| 74 | 102 | 165 |

**SOLUTION CODE:**

# -\*- coding: utf-8 -\*-

"""

Created on Mon Nov 13 10:28:40 2017

@author: umerm

"""

from queue import PriorityQueue

heuristics={'2':3,

'5':2,

'45':1,

'74':2,

'80':3,

'102':1,

'145':2,

'150':4,

'165':0

};

def A\_Star\_Search(Graph,Start\_Node,Goal):

PQ=PriorityQueue();

PQ.put((0,Start\_Node));

came\_From={};

cost\_so\_Far={};

came\_From[Start\_Node]=None;

cost\_so\_Far[Start\_Node]=0;

while not PQ.empty():

current\_Node=PQ.get();

for neighbours in Graph[current\_Node[1]]:

for key in neighbours.keys():

new\_Cost=cost\_so\_Far[current\_Node[1]]+neighbours[key];

if key not in cost\_so\_Far or new\_Cost < cost\_so\_Far[key]:

cost\_so\_Far[key]=new\_Cost;

priority=new\_Cost+heuristics[key];

PQ.put((priority,key));

came\_From[key]=current\_Node[1];

return re\_ContructPath(came\_From,Start\_Node,Goal);

def re\_ContructPath(came\_From,start,goal):

current=goal;

path=[];

while current is not start:

path = path + [current];

current=came\_From[current];

path = path + [start];

path\_reverse=path[::-1];

return path\_reverse

def main():

Graph\_Image={'2':[{'5':3},{'145':143},{'150':148}],

'5':[{'2':3},{'45':43}],

'45':[{'5':40},{'145':100},{'165':120}],

'74':[{'80':6},{'102':28}],

'80':[{'74':6},{'145':65},{'150':70}],

'102':[{'74':28},{'145':43},{'165':63}],

'145':[{'2':143},{'45':100},{'80':65},{'102':43}],

'150':[{'2':148},{'80':70}],

'165':[{'45':120},{'102':63}]

}

print(A\_Star\_Search(Graph\_Image,'150','165'));

main();

**OUTPUT:**

