

SRM INSTITUTE OF SCIENCE & TECHNOLOGY DEPARTMENT OF NETWORKING & COMMUNICATIONS **18CSC305J-ARTIFICIAL INTELLIGENCE**

SEMESTER – 6 BATCH-2

|  |  |
| --- | --- |
| **REGISTRATION NUMBER** | **RA1911028010018** |
| **NAME** | VAIBHAV SHARMA |

# B.Tech- CSE / CC, Third Year (Section: H2)

**Faculty In charge: Dr. S. Prabakeran, B.Tech, M.E, PhD**

**Assistant Professor**

**School of Computing - Department of**

**Networking and Communications**

**Year 2021-2022 / Even Semester**

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ex No** | **DATE** | **Title** | **Page No** | **Marks** |
| 1 | 15/02/22 | Developing Best first search and A\* Algorithm for real world problems |  |  |

**Date : 15-02-2022**

# Best First Search (Informed Search)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

## Algorithm :

1) Create an empty PriorityQueue

PriorityQueue **pq**;

2) Insert "start" in pq.

pq.insert(start)

3) Until PriorityQueue is empty

u = PriorityQueue.DeleteMin

If u is the goal

Exit

Else

Foreach neighbor v of u

If v "Unvisited"

Mark v "Visited"

pq.insert(v)

Mark u "Examined"

End procedure

## A\* Algorithm

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

**Algorithm**

* Make an open list containing starting node
  + If it reaches the destination node :
  + Make a closed empty list
  + If it does not reach the destination node, then consider a node with the lowest f-score in the open list

We are finished

* Else :

Put the current node in the list and check its neighbors

* For each neighbor of the current node :
  + If the neighbor has a lower g value than the current node and is in the closed list:

Replace neighbor with this new node as the neighbor’s parent

* Else If (current g is lower and neighbor is in the open list):

Replace neighbor with the lower g value and change the neighbor’s parent to the current node.

* Else If the neighbor is not in both lists:

Add it to the open list and set its g

**Tool :** AWS and Python 3.9.0

## Programming code :

## A-star

#ex 5 a star

class Graph:

    def \_\_init\_\_(self, graph\_dict=None, directed=True):

        self.graph\_dict = graph\_dict or {}

        self.directed = directed

        if not directed:

            self.make\_undirected()

    def make\_undirected(self):

        for a in list(self.graph\_dict.keys()):

            for (b, dist) in self.graph\_dict[a].items():

                self.graph\_dict.setdefault(b, {})[a] = dist

    def connect(self, A, B, distance=1):

        self.graph\_dict.setdefault(A, {})[B] = distance

        if not self.directed:

            self.graph\_dict.setdefault(B, {})[A] = distance

    def get(self, a, b=None):

        links = self.graph\_dict.setdefault(a, {})

        if b is None:

            return links

        else:

            return links.get(b)

    # return list of nodes in the graph

    def nodes(self):

        s1 = set([k for k in self.graph\_dict.keys()])

        s2 = set([k2 for v in self.graph\_dict.values() for k2, v2 in v.items()])

        nodes = s1.union(s2)

        return list(nodes)

# node class

class Node:

    # init class

    def \_\_init\_\_(self, name:str, parent:str):

        self.name = name

        self.parent = parent

        self.g = 0 # distance to start node

        self.h = 0 # distance to goal node

        self.f = 0 # total cost

    # compare nodes

    def \_\_eq\_\_(self, other):

        return self.name == other.name

    # sort nodes

    def \_\_lt\_\_(self, other):

         return self.f < other.f

    # print node

    def \_\_repr\_\_(self):

        return ('({0},{1})'.format(self.name, self.f))

# A\* search

def astar\_search(graph, heuristics, start, end):

    # lists for open nodes and closed nodes

    open = []

    closed = []

    # a start node and an goal node

    start\_node = Node(start, None)

    goal\_node = Node(end, None)

    # add start node

    open.append(start\_node)

    # loop until the open list is empty

    while len(open) > 0:

        open.sort()                                 # sort open list to get the node with the lowest cost first

        current\_node = open.pop(0)                  # get node with the lowest cost

        closed.append(current\_node)                 # add current node to the closed list

        # check if we have reached the goal, return the path

        if current\_node == goal\_node:

            path = []

            while current\_node != start\_node:

                path.append(current\_node.name + ': ' + str(current\_node.g))

                current\_node = current\_node.parent

            path.append(start\_node.name + ': ' + str(start\_node.g))

            return path[::-1]

        neighbors = graph.get(current\_node.name)    # get neighbours

        # loop neighbors

        for key, value in neighbors.items():

            neighbor = Node(key, current\_node)      # create neighbor node

            if(neighbor in closed):                 # check if the neighbor is in the closed list

                continue

            # calculate full path cost

            neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

            neighbor.h = heuristics.get(neighbor.name)

            neighbor.f = neighbor.g + neighbor.h

            # check if neighbor is in open list and if it has a lower f value

            if(add\_to\_open(open, neighbor) == True):

                # everything is green, add neighbor to open list

                open.append(neighbor)

    # return None, no path is found

    return None

# check if a neighbor should be added to open list

def add\_to\_open(open, neighbor):

    for node in open:

        if (neighbor == node and neighbor.f > node.f):

            return False

    return True

# create a graph

graph = Graph() # user-based input for edges will be updated in the upcoming days

# create graph connections (Actual distance)

graph.connect('Frankfurt', 'Wurzburg', 111)

graph.connect('Frankfurt', 'Mannheim', 85)

graph.connect('Wurzburg', 'Nurnberg', 104)

graph.connect('Wurzburg', 'Stuttgart', 140)

graph.connect('Wurzburg', 'Ulm', 183)

graph.connect('Mannheim', 'Nurnberg', 230)

graph.connect('Mannheim', 'Karlsruhe', 67)

graph.connect('Karlsruhe', 'Basel', 191)

graph.connect('Karlsruhe', 'Stuttgart', 64)

graph.connect('Nurnberg', 'Ulm', 171)

graph.connect('Nurnberg', 'Munchen', 170)

graph.connect('Nurnberg', 'Passau', 220)

graph.connect('Stuttgart', 'Ulm', 107)

graph.connect('Basel', 'Bern', 91)

graph.connect('Basel', 'Zurich', 85)

graph.connect('Bern', 'Zurich', 120)

graph.connect('Zurich', 'Memmingen', 184)

graph.connect('Memmingen', 'Ulm', 55)

graph.connect('Memmingen', 'Munchen', 115)

graph.connect('Munchen', 'Ulm', 123)

graph.connect('Munchen', 'Passau', 189)

graph.connect('Munchen', 'Rosenheim', 59)

graph.connect('Rosenheim', 'Salzburg', 81)

graph.connect('Passau', 'Linz', 102)

graph.connect('Salzburg', 'Linz', 126)

# make graph undirected, create symmetric connections

graph.make\_undirected()

# create heuristics (straight-line distance, air-travel distance)

heuristics = {}

heuristics['Basel'] = 204

heuristics['Bern'] = 247

heuristics['Frankfurt'] = 215

heuristics['Karlsruhe'] = 137

heuristics['Linz'] = 318

heuristics['Mannheim'] = 164

heuristics['Munchen'] = 120

heuristics['Memmingen'] = 47

heuristics['Nurnberg'] = 132

heuristics['Passau'] = 257

heuristics['Rosenheim'] = 168

heuristics['Stuttgart'] = 75

heuristics['Salzburg'] = 236

heuristics['Wurzburg'] = 153

heuristics['Zurich'] = 157

heuristics['Ulm'] = 0

# run the search algorithm

path = astar\_search(graph, heuristics, 'Frankfurt', 'Ulm')

print("Path:", path)

**Best First Search**

#ex 5 bfs best first search

from queue import PriorityQueue

v = 5

graph = [[] for i in range(v)]

def best\_first\_search(source, target, n):

  visited = [0] \* n

  visited[0] = True

  pq = PriorityQueue()

  pq.put((0, source))

  while pq.empty() == False:

    u = pq.get()[1]

    print(u, end=" ")

    if u == target:

      break

    for v, c in graph[u]:

      if visited[v] == False:

        visited[v] = True

        pq.put((c, v))

  print()

def addedge(x, y, cost):

  graph[x].append((y, cost))

  graph[y].append((x, cost))

addedge(0, 1, 5)

addedge(0, 2, 1)

addedge(2, 3, 2)

addedge(1, 4, 1)

addedge(3, 4, 2)

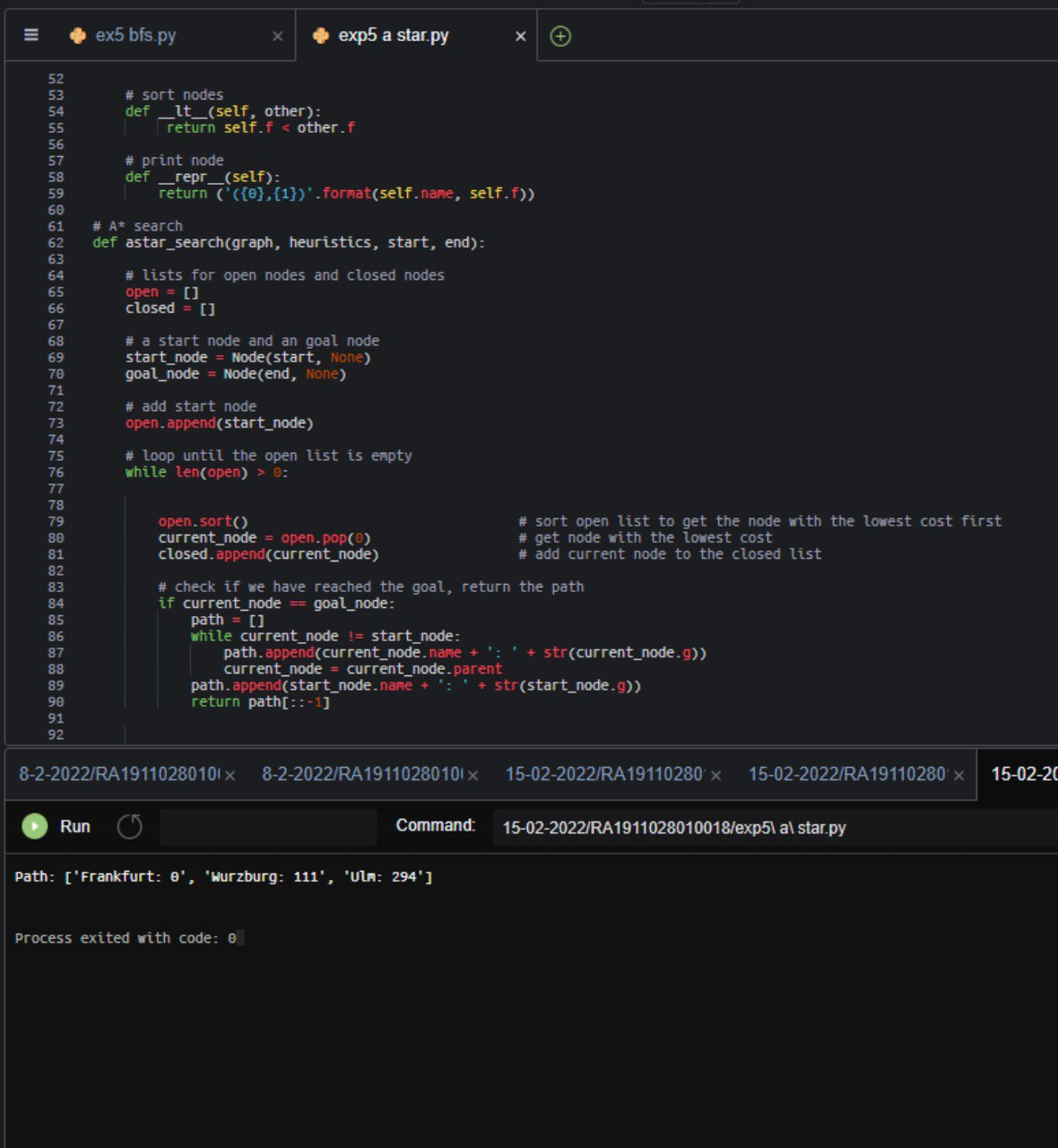
source = 0

target = 4

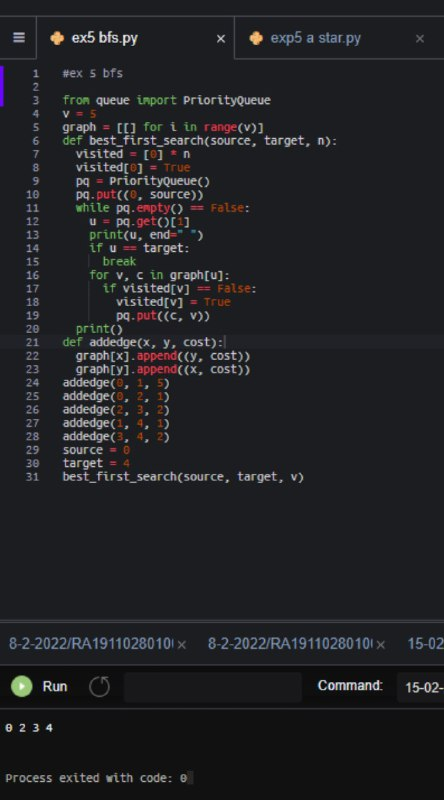
best\_first\_search(source, target, v)

## Output screen shots :

## A\*



**BFS**



**Result :** A\* and Best first search algorithms were implemented successfully.