SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF NETWORKING & COMMUNICATIONS

18CSC305J-ARTIFICIAL INTELLIGENCE

SEMESTER – 6

BATCH-1

REGISTRATION NUMBER	**RA1911003011013**
NAME	**Shivank Arora**

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	21/02/22	**Developing Best first search and A* Algorithm for real world problems**		

Exercise: 5

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 : VS Code and Python 3.9.0

Programming code :

A-star

```
# graph class
class Graph:

    # init class
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    # create undirected graph by adding symmetric edges
    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

    # add link from A and B of given distance, and also add the inverse link if the graph is
undirected
    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

    # get neighbors or a neighbor
    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()
```

```
# create graph connections (Actual distance)
graph.connect('S', 'A', 2)
graph.connect('S', 'G', 20)
graph.connect('A', 'C', 7)
graph.connect('C', 'G', 8)
graph.connect('C', 'D', 9)
graph.connect('D', 'G', 10)
# make graph undirected, create symmetric connections
graph.make_undirected()
# create heuristics (straight-line distance, air-travel distance)
heuristics = {}
heuristics['A'] = 5
heuristics['C'] = 8
heuristics['G'] = 7
heuristics['D'] = 6
heuristics['S'] = 9
# run the search algorithm
path = astar_search(graph, heuristics, 'S', 'G')
print("Path:", path)
```

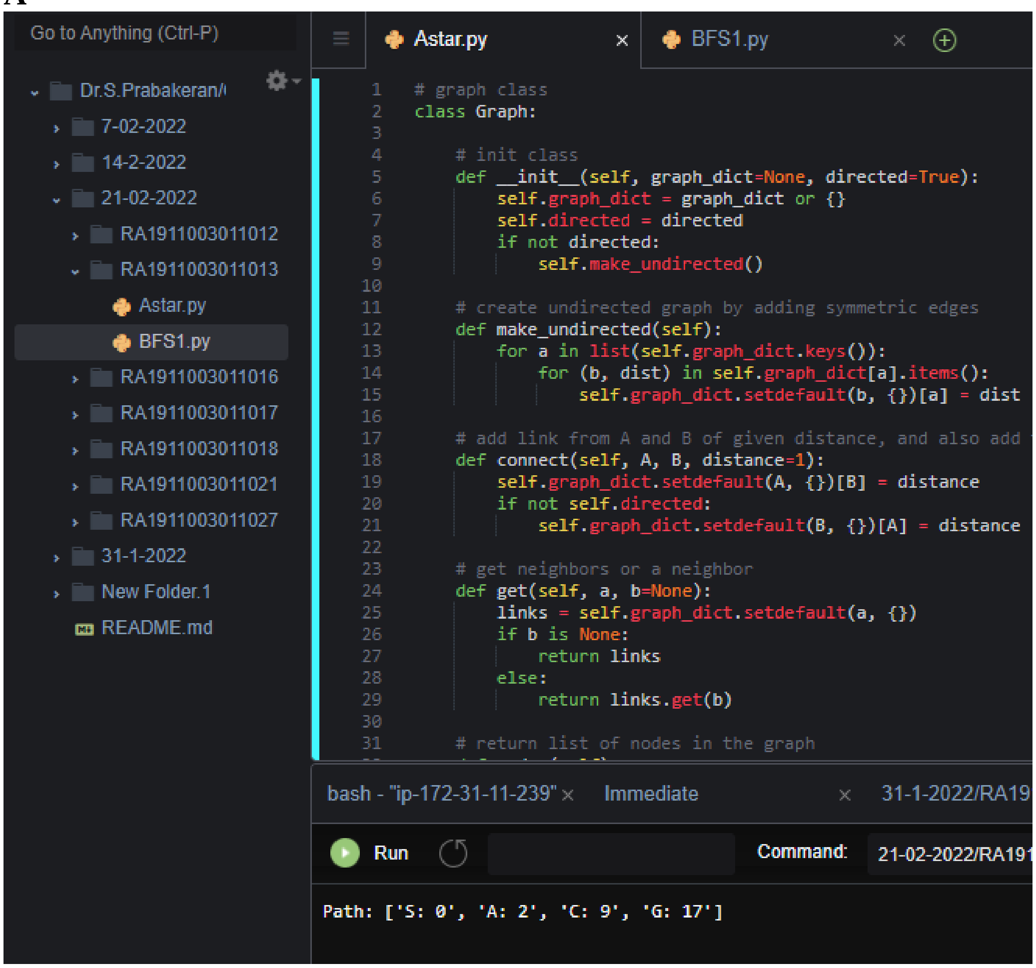
Best First Search

```
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))
```

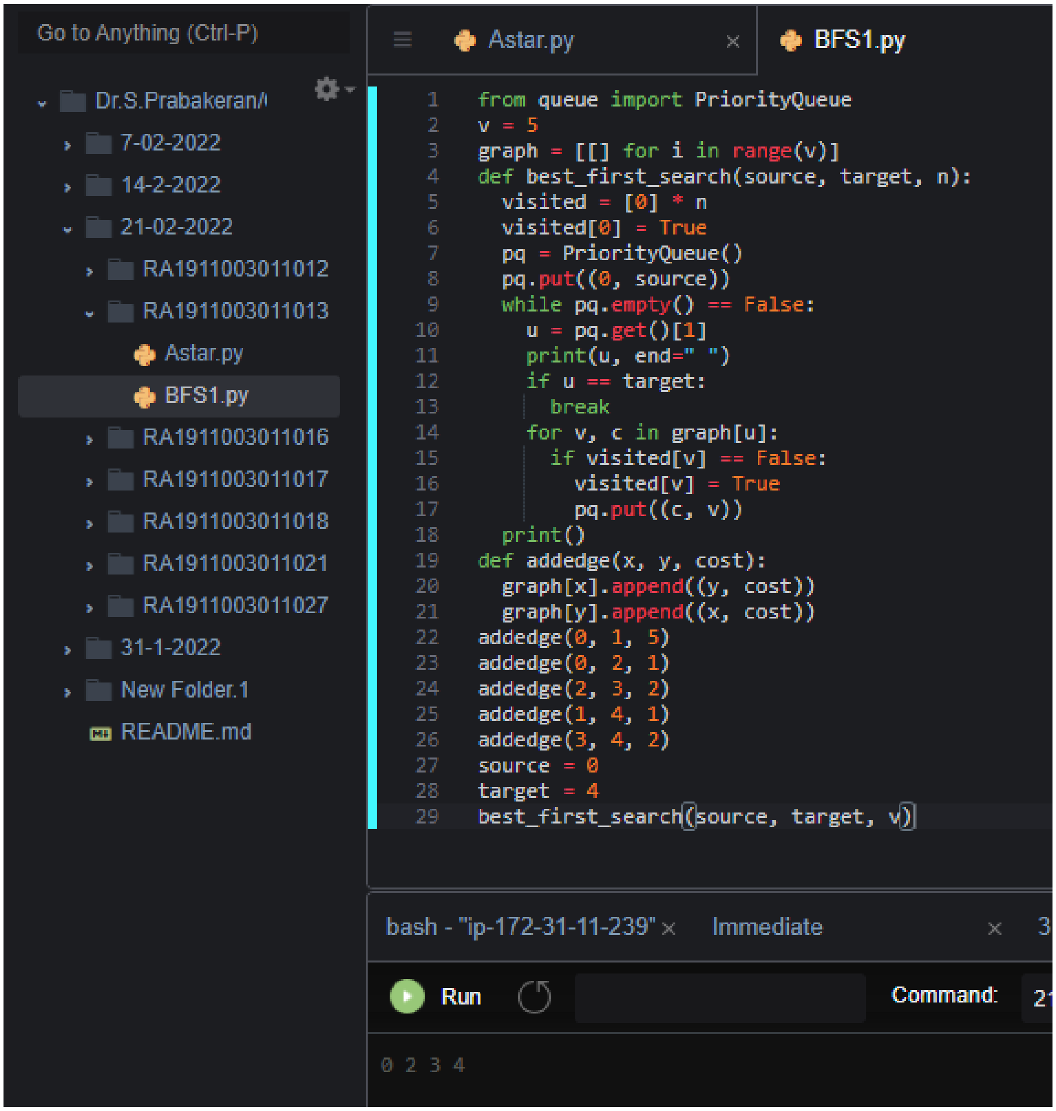
```
  graph[y].append((x, cost))
addedge(0, 1, 5)
addedge(0, 2, 1)
addedge(2, 3, 2)
addedge(1, 4, 1)
addedge(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)
```

Output screen shots :

A*-



BFS-



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