**Problem Statement –** Implement A star Algorithm for any game search problem.

# A\* Search Algorithm

#

# let openList equal empty list of nodes

# let closedList equal empty list of nodes

# put startNode on the openList (leave it's f at zero)

# while openList is not empty

# let currentNode equal the node with the least f value

# remove currentNode from the openList

# add currentNode to the closedList

# if currentNode is the goal

# You've found the exit!

# let children of the currentNode equal the adjacent nodes

# for each child in the children

# if child is in the closedList

# continue to beginning of for loop

# child.g = currentNode.g + distance b/w child and current

# child.h = distance from child to end

# child.f = child.g + child.h

# if child.position is in the openList's nodes positions

# if child.g is higher than the openList node's g

# continue to beginning of for loop

# add the child to the openList

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

# heuristic function with distances from the current node to the goal node

def h(self, n):

H = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

# open\_list is a list of nodes which have been visited, but who's neighbors

# haven't all been inspected, starts off with the start node

# closed\_list is a list of nodes which have been visited

# and who's neighbors have been inspected

open\_list = set([start\_node])

closed\_list = set([])

# g contains current distances from start\_node to all other nodes

# the default value (if it's not found in the map) is +infinity

g = {}

g[start\_node] = 0

# parents contains an adjacency map of all nodes

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# find a node with the lowest value of f() - evaluation function

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

# for all neighbors of the current node do

for (m, weight) in self.get\_neighbors(n):

# if the current node isn't in both open\_list and closed\_list

# add it to open\_list and note n as it's parent

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m

# and if it is, update parent data and g data

# and if the node was in the closed\_list, move it to open\_list

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjac\_lis = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1), ('G', 9)],

'C': None,

'D': [('G', 1)],

'E': [('D', 6)]

}

graph = Graph(adjac\_lis)

graph.a\_star\_algorithm('A', 'G')

Output:-

Path found: ['A', 'E', 'D', 'G']