# TASK:3

Implementation of **A \* Algorithm** to find the optimal path using Python by following constraints.

# 3(A) A\* Algorithm

**Aim :** To implement of A \* Algorithm to find the optimal path using Jupiter notebook.

# Algorithm:

**Step 1:** start

**Step 2:** Place the starting node into open and find its f(n) [start node] value.

**Step 3:** Remove the node from OPEN, having the smallest f(n) value, if it is x goal node, then stop and return to success.

**Step 4:** Else remove the node from OPEN, and find all its successors.

**Step 5:** Find the f(n) value of all the successors, Place them into OPEN and place the removed node into close

**Step 6:** Go to step 2.

**Step 7:** Exit.

# Program :

# def aStarAlgo(start\_node, stop\_node):

# open\_set = set([start\_node])

# closed\_set = set()

# g = {} # store distance from starting node

# parents = {} # store parent nodes

# # distance of starting node from itself is zero

# g[start\_node] = 0

# # start\_node is the root node, so it has no parent

# parents[start\_node] = start\_node

# while len(open\_set) > 0:

# n = None

# # node with the lowest f() = g(n) + h(n)

# for v in open\_set:

# if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

# n = v

# if n is None:

# print('Path does not exist!')

# return None

# # if goal node is reached

# if n == stop\_node:

# path = []

# while parents[n] != n:

# path.append(n)

# n = parents[n]

# path.append(start\_node)

# path.reverse()

# print('Path found:', path)

# return path

# # check neighbors

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

# if m not in open\_set and m not in closed\_set:

# open\_set.add(m)

# parents[m] = n

# g[m] = g[n] + weight

# else:

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

# # update g(m)

# g[m] = g[n] + weight

# parents[m] = n

# # move m to open\_set if it was in closed\_set

# if m in closed\_set:

# closed\_set.remove(m)

# open\_set.add(m)

# # remove n from open\_set, add to closed\_set

# open\_set.remove(n)

# closed\_set.add(n)

# print('Path does not exist!')

# return None

# # neighbors with weights

# def get\_neighbors(v):

# if v in Graph\_nodes:

# return Graph\_nodes[v]

# else:

# return None

# # heuristic distances

# def heuristic(n):

# h\_dist = {

# 'A': 11,

# 'B': 6,

# 'C': 5,

# 'D': 7,

# 'E': 3,

# 'F': 6,

# 'G': 5,

# 'H': 3,

# 'I': 1,

# 'J': 0

# }

# return h\_dist[n]

# # graph definition

# Graph\_nodes = {

# 'A': [('B', 6), ('F', 3)],

# 'B': [('A', 6), ('C', 3), ('D', 2)],

# 'C': [('B', 3), ('D', 1), ('E', 5)],

# 'D': [('B', 2), ('C', 1), ('E', 8)],

# 'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

# 'F': [('A', 3), ('G', 1), ('H', 7)],

# 'G': [('F', 1), ('I', 3)],

# 'H': [('F', 7), ('I', 2)],

# 'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

# }

# print("Following is the A\* Algorithm:")

# aStarAlgo('A', 'J')

# Output:

# A screenshot of a computer AI-generated content may be incorrect.

**Result:**

Thus the Implementation of A \* Algorithm to find the optimal path using Python Was successfully executed and output was verified.

# 3(B) – Simplified A\* Algorithm.

**Aim:** To implement the simplified A\*Algorithm using Jupiter notebook.

# Algorithm:

**Step 1 :** start.

**Step 2:** place the starting node into open and find its f(n) value

**Step 3:** Remove the node from OPEN , having the smallest f(n) value, if it is a goal node , then stop and return to success.

**Step 4:** else remove the node from OPEN, and find all its successors

**Step 5:**Find the f(n) value of all the successors, Place them into OPEN and place the removed node into close

**Step 6:** Go to step 2.

**Step 7:** Exit.

# Program:

# def aStarAlgo(start\_node, stop\_node):

# open\_set = set([start\_node])

# closed\_set = set()

# g = {} # store distance from starting node

# parents = {} # store parent nodes

# # distance of starting node from itself is zero

# g[start\_node] = 0

# parents[start\_node] = start\_node

# while len(open\_set) > 0:

# n = None

# # node with the lowest f() = g(n) + h(n)

# for v in open\_set:

# if n is None or g[v] + heuristic(v) < g[n] + heuristic(n):

# n = v

# if n is None:

# print('Path does not exist!')

# return None

# # if goal node is reached

# if n == stop\_node:

# path = []

# while parents[n] != n:

# path.append(n)

# n = parents[n]

# path.append(start\_node)

# path.reverse()

# print('Path found:', path)

# return path

# # explore neighbors

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

# if m not in open\_set and m not in closed\_set:

# open\_set.add(m)

# parents[m] = n

# g[m] = g[n] + weight

# else:

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

# # update g(m)

# g[m] = g[n] + weight

# parents[m] = n

# if m in closed\_set:

# closed\_set.remove(m)

# open\_set.add(m)

# open\_set.remove(n)

# closed\_set.add(n)

# print('Path does not exist!')

# return None

# # get neighbors

# def get\_neighbors(v):

# if v in Graph\_nodes:

# return Graph\_nodes[v]

# else:

# return None

# # heuristic values

# def heuristic(n):

# h\_dist = {

# 'A': 11,

# 'B': 6,

# 'C': 99,

# 'D': 1,

# 'E': 7,

# 'G': 0

# }

# return h\_dist[n]

# # graph definition

# Graph\_nodes = {

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

# 'B': [('A', 2), ('C', 1), ('G', 9)],

# 'C': [('B', 1)],

# 'D': [('E', 6), ('G', 1)],

# 'E': [('A', 3), ('D', 6)],

# 'G': [('B', 9), ('D', 1)]

# }

# print("Following is the A\* Algorithm:")

# aStarAlgo('A', 'G')

# Output:

# A screenshot of a computer AI-generated content may be incorrect.

# Result:

Thus the implementation of the simplified A\*Algorithm using Jupiter notebook was successfully executed and output was verified.