**ST.FRANCIS INSTITUTE OF TECHNOLOGY**

Mount Poinsur, S.V.P. Road, Borivli (West), Mumbai - 400103

**Computer Engineering Department**

**Academic Year:** 2021-2022 **Class/Branch**: BE CMPN

**Subject:** CSC 703 Artificial Intelligence & Soft Computing **Semester**: VII

**Experiment No. 4 A\* Algorithm**

**Aim:** To learn the working of A\*Algorithm and apply it in state space search to achieve the solution goals of well defined problems.

**Theory:**

A-star (also referred as A\*) is one of the most successful search algorithms to find the shortest path between nodes of graphs. It is an informed search algorithm , as it uses information about path cost and also heuristics to find the solution.

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

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

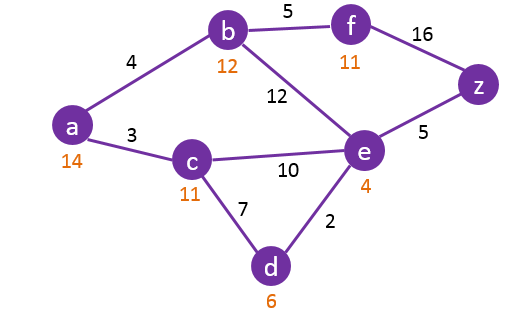
g(n) the cost (so far) to reach the node

h(n) estimated cost to get from the node to the goal

f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

**Experiment:**

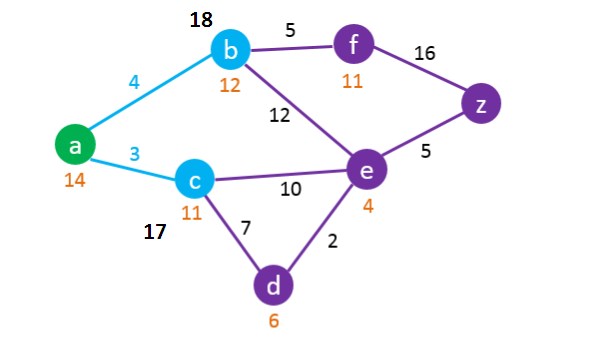
The aim of A\* algorithm is to traverse the graph from start node A to end node Z.. Stack is used in the implementation of this algorithm. Consider how A\* Algorithm reaches the destination node based on the information given with respect to the following graph:

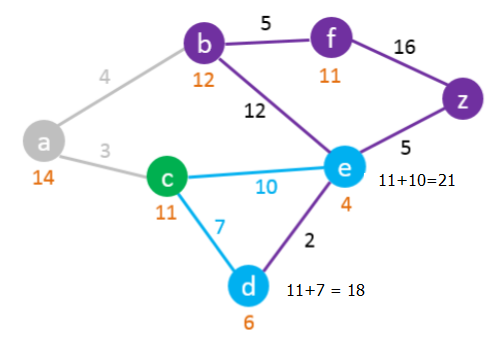


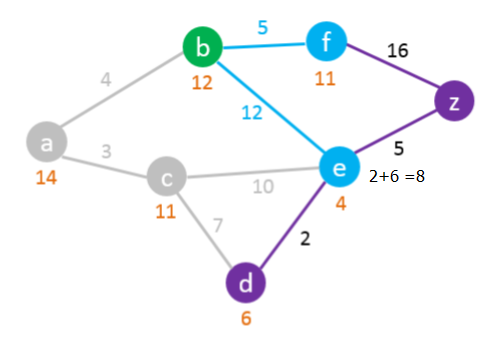
**Algorithmic Steps**

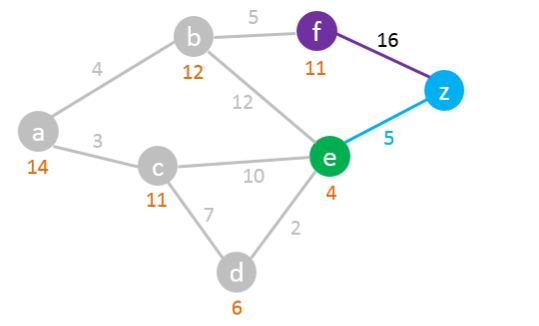
1. **Step 1**: Push the root node in the Stack.
2. **Step 2**: Compute f(n) to the connected node
3. **Step 3**:Select the node with the lowest value of f(n) and move it to stack.
4. **Step 4**: If the removed node has unvisited child nodes, mark them as visited and consider the new node and check if it is the end node. If the node is not end node treat it as the root node and follow Step 2, else stop to retrieve path from stack .

Example:









We found the shortest path from A to Z.

Read the path from Z to A using the previous node column:

Z > E > D > C > A

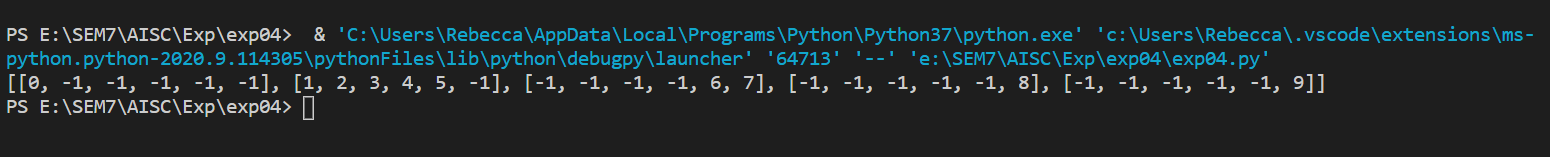
So the Shortest Path is: A – C – D – E – Z with a length of 17

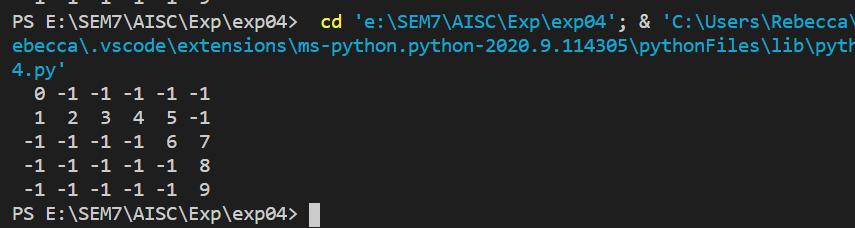
**Experiment Exercise:** Implement A\* Algorithm

**Program:**

|  |
| --- |
| import numpy as np  class Node:      """          A node class for A\* Pathfinding          parent is parent of the current Node          position is current position of the Node in the maze          g is cost from start to current Node          h is heuristic based estimated cost for current Node to end Node          f is total cost of present node i.e. :  f = g + h      """      def \_\_init\_\_(self, parent=None, position=None):          self.parent = parent          self.position = position          self.g = 0          self.h = 0          self.f = 0      def \_\_eq\_\_(self, other):          return self.position == other.position  #This function return the path of the search  def return\_path(current\_node,maze):      path = []      no\_rows, no\_columns = np.shape(maze)      # here we create the initialized result maze with -1 in every position      result = [[-1 for i in range(no\_columns)] for j in range(no\_rows)]      current = current\_node      while current is not None:          path.append(current.position)          current = current.parent      # Return reversed path as we need to show from start to end path      path = path[::-1]      start\_value = 0      # we update the path of start to end found by A-star serch with every step incremented by 1      for i in range(len(path)):          result[path[i][0]][path[i][1]] = start\_value          start\_value += 1      return result  def search(maze, cost, start, end):      """          Returns a list of tuples as a path from the given start to the given end in the given maze          :param maze:          :param cost          :param start:          :param end:          :return:      """      # Create start and end node with initized values for g, h and f      start\_node = Node(None, tuple(start))      start\_node.g = start\_node.h = start\_node.f = 0      end\_node = Node(None, tuple(end))      end\_node.g = end\_node.h = end\_node.f = 0      # Initialize both yet\_to\_visit and visited list      # in this list we will put all node that are yet\_to\_visit for exploration.      # From here we will find the lowest cost node to expand next      yet\_to\_visit\_list = []      # in this list we will put all node those already explored so that we don't explore it again      visited\_list = []        # Add the start node      yet\_to\_visit\_list.append(start\_node)        # Adding a stop condition. This is to avoid any infinite loop and stop      # execution after some reasonable number of steps      outer\_iterations = 0      max\_iterations = (len(maze) // 2) \*\* 10      # what squares do we search . serarch movement is left-right-top-bottom      #(4 movements) from every positon      move  =  [[-1, 0 ], # go up                [ 0, -1], # go left                [ 1, 0 ], # go down                [ 0, 1 ]] # go right      """          1) We first get the current node by comparing all f cost and selecting the lowest cost node for further expansion          2) Check max iteration reached or not . Set a message and stop execution          3) Remove the selected node from yet\_to\_visit list and add this node to visited list          4) Perofmr Goal test and return the path else perform below steps          5) For selected node find out all children (use move to find children)              a) get the current postion for the selected node (this becomes parent node for the children)              b) check if a valid position exist (boundary will make few nodes invalid)              c) if any node is a wall then ignore that              d) add to valid children node list for the selected parent                For all the children node                  a) if child in visited list then ignore it and try next node                  b) calculate child node g, h and f values                  c) if child in yet\_to\_visit list then ignore it                  d) else move the child to yet\_to\_visit list      """      #find maze has got how many rows and columns      no\_rows, no\_columns = np.shape(maze)        # Loop until you find the end        while len(yet\_to\_visit\_list) > 0:            # Every time any node is referred from yet\_to\_visit list, counter of limit operation incremented          outer\_iterations += 1            # Get the current node          current\_node = yet\_to\_visit\_list[0]          current\_index = 0          for index, item in enumerate(yet\_to\_visit\_list):              if item.f < current\_node.f:                  current\_node = item                  current\_index = index            # if we hit this point return the path such as it may be no solution or          # computation cost is too high          if outer\_iterations > max\_iterations:              print ("giving up on pathfinding too many iterations")              return return\_path(current\_node,maze)          # Pop current node out off yet\_to\_visit list, add to visited list          yet\_to\_visit\_list.pop(current\_index)          visited\_list.append(current\_node)          # test if goal is reached or not, if yes then return the path          if current\_node == end\_node:              return return\_path(current\_node,maze)          # Generate children from all adjacent squares          children = []          for new\_position in move:              # Get node position              node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])              # Make sure within range (check if within maze boundary)              if (node\_position[0] > (no\_rows - 1) or                  node\_position[0] < 0 or                  node\_position[1] > (no\_columns -1) or                  node\_position[1] < 0):                  continue              # Make sure walkable terrain              if maze[node\_position[0]][node\_position[1]] != 0:                  continue              # Create new node              new\_node = Node(current\_node, node\_position)              # Append              children.append(new\_node)          # Loop through children          for child in children:                # Child is on the visited list (search entire visited list)              if len([visited\_child for visited\_child in visited\_list if visited\_child == child]) > 0:                  continue              # Create the f, g, and h values              child.g = current\_node.g + cost              ## Heuristic costs calculated here, this is using eucledian distance              child.h = (((child.position[0] - end\_node.position[0]) \*\* 2) +                         ((child.position[1] - end\_node.position[1]) \*\* 2))              child.f = child.g + child.h              # Child is already in the yet\_to\_visit list and g cost is already lower              if len([i for i in yet\_to\_visit\_list if child == i and child.g > i.g]) > 0:                  continue              # Add the child to the yet\_to\_visit list              yet\_to\_visit\_list.append(child)  if \_\_name\_\_ == '\_\_main\_\_':      maze = [[0, 1, 0, 0, 0, 0],              [0, 0, 0, 0, 0, 0],              [0, 1, 0, 1, 0, 0],              [0, 1, 0, 0, 1, 0],              [0, 0, 0, 0, 1, 0]]        start = [0, 0] # starting position      end = [4,5] # ending position      cost = 1 # cost per movement      path = search(maze,cost, start, end)      print(path)  print('\n'.join([''.join(["{:" ">3d}".format(item) for item in row])        for row in path])) |

**OUTPUT:**



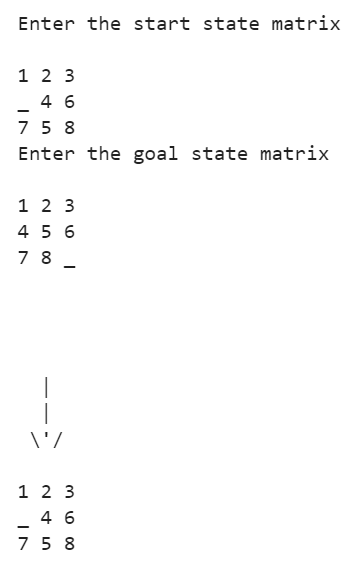
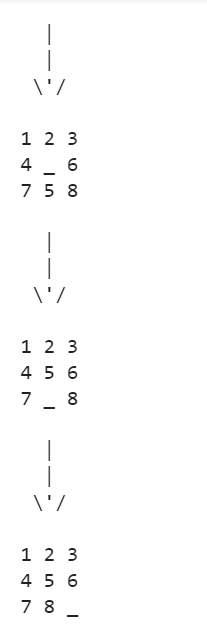


**Post Experiment Exercise:**

1. Implement A\* Algorithm to solve 8-puzzle problems using any programming language.

|  |
| --- |
| class Node:      def \_\_init\_\_(self,data,level,fval):          """ Initialize the node with the data, level of the node and the calculated fvalue """          self.data = data          self.level = level          self.fval = fval      def generate\_child(self):          """ Generate child nodes from the given node by moving the blank space              either in the four directions {up,down,left,right} """          x,y = self.find(self.data,'\_')          """ val\_list contains position values for moving the blank space in either of              the 4 directions [up,down,left,right] respectively. """          val\_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]          children = []          for i in val\_list:              child = self.shuffle(self.data,x,y,i[0],i[1])              if child is not None:                  child\_node = Node(child,self.level+1,0)                  children.append(child\_node)          return children        def shuffle(self,puz,x1,y1,x2,y2):          """ Move the blank space in the given direction and if the position value are out              of limits the return None """          if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):              temp\_puz = []              temp\_puz = self.copy(puz)              temp = temp\_puz[x2][y2]              temp\_puz[x2][y2] = temp\_puz[x1][y1]              temp\_puz[x1][y1] = temp              return temp\_puz          else:              return None        def copy(self,root):          """ Copy function to create a similar matrix of the given node"""          temp = []          for i in root:              t = []              for j in i:                  t.append(j)              temp.append(t)          return temp        def find(self,puz,x):          """ Specifically used to find the position of the blank space """          for i in range(0,len(self.data)):              for j in range(0,len(self.data)):                  if puz[i][j] == x:                      return i,j  class Puzzle:      def \_\_init\_\_(self,size):          """ Initialize the puzzle size by the specified size,open and closed lists to empty """          self.n = size          self.open = []          self.closed = []      def accept(self):          """ Accepts the puzzle from the user """          puz = []          for i in range(0,self.n):              temp = input().split(" ")              puz.append(temp)          return puz      def f(self,start,goal):          """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """          return self.h(start.data,goal)+start.level      def h(self,start,goal):          """ Calculates the different between the given puzzles """          temp = 0          for i in range(0,self.n):              for j in range(0,self.n):                  if start[i][j] != goal[i][j] and start[i][j] != '\_':                      temp += 1          return temp        def process(self):          """ Accept Start and Goal Puzzle state"""          print("Enter the start state matrix \n")          start = self.accept()          print("Enter the goal state matrix \n")          goal = self.accept()          start = Node(start,0,0)          start.fval = self.f(start,goal)          """ Put the start node in the open list"""          self.open.append(start)          print("\n\n")          while True:              cur = self.open[0]              print("")              print("  | ")              print("  | ")              print(" \\\'/ \n")              for i in cur.data:                  for j in i:                      print(j,end=" ")                  print("")              """ If the difference between current and goal node is 0 we have reached the goal node"""              if(self.h(cur.data,goal) == 0):                  break              for i in cur.generate\_child():                  i.fval = self.f(i,goal)                  self.open.append(i)              self.closed.append(cur)              del self.open[0]              """ sort the opne list based on f value """              self.open.sort(key = lambda x:x.fval,reverse=False)  puz = Puzzle(3)  puz.process() |

**OUTPUT:**

**Conclusion:** Problem Solving using A\* has been explored and implemented using Python.