EXPERIMENT:-1

Objective:- Write a program to implement Breadth First Search Algorithm.

Description:- Breadth-First Search (BFS) is an algorithm used for traversing graphs or trees. Traversing means visiting each node of the graph. Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree. BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. The only difference is that the graph may contain cycles, so we may traverse to the same node again.

Algorithm:-

- Start by putting any one of the graph's vertices at the back of the queue.
- Now take the front item of the queue and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
- Keep continuing steps two and three till the queue is empty.

Many times, a graph may contain two different disconnected parts and therefore to make sure that we have visited every vertex, we can also run the BFS algorithm at every node. The time complexity of the Breadth first Search algorithm is in the form of O(V+E), where V is the representation of the number of nodes and E is the number of edges. Also, the space complexity of the BFS algorithm is O(V).

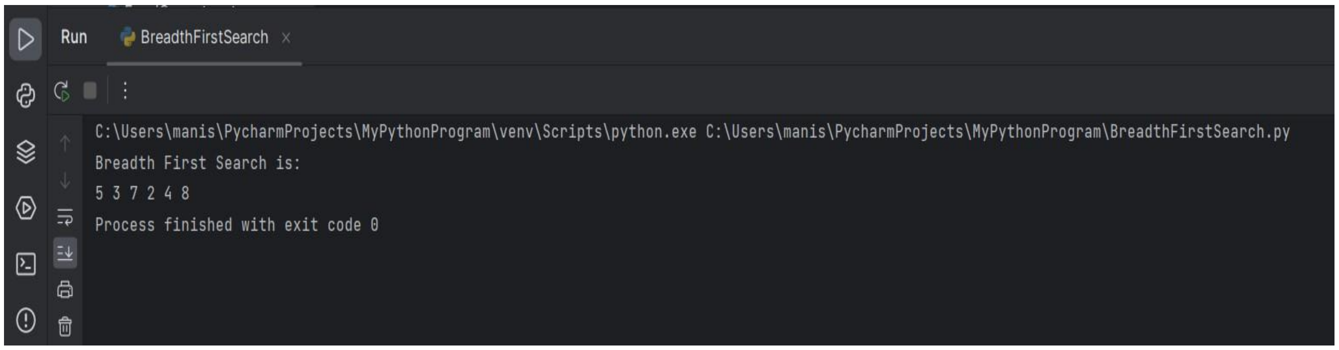
Pseudo code:-

```
○ create a queue Q
○ mark v as visited and put v into Q while Q is non-empty
○ remove the head u of Q
○ mark and enqueue all (unvisited) neighbours of u
```

Program:-

```
graph = {
  '5': ['3', '7'],
  '3': ['2', '4'],
  '7': ['8'],
  '2': [],
  '4': ['8'],
  '8': []
}
visited = []
queue = []
def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
        m = queue.pop(0)
        print(m, end=" ")
        for neighbour in graph[m]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
print("Breadth First Search is:")
bfs(visited, graph, '5')
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\BreadthFirstSearch.py
Breadth First Search is:
5 3 7 2 4 8
Process finished with exit code 0
```

EXPERIMENT:-2

Objective:- Write a program to implement Depth First Search Algorithm.

Description:- The Depth-First Search is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

Algorithm:-

- We will start by putting any one of the graph's vertex on top of the stack.
- After that take the top item of the stack and add it to the visited list of the vertex.
- Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertexes to the top of the stack.
- Lastly, keep repeating steps 2 and 3 until the stack is empty.

The time complexity of the Depth-First Search algorithm is represented within the sort of O (V + E), where V is that the number of nodes and E is that the number of edges.The space complexity of the algorithm is O(V).

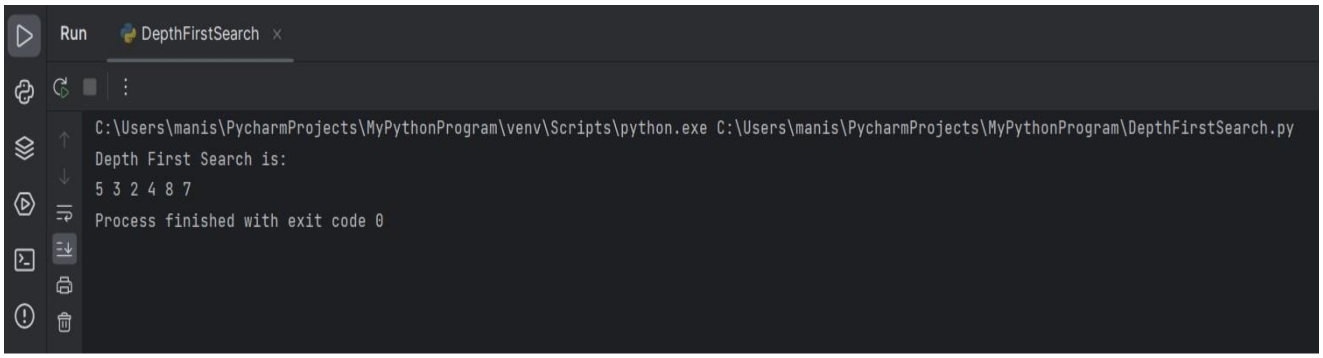
Pseudo Code:-

```
DFS(G, u)
u.visited = true
for each v ∈ G.Adj[u] if v.visited == false
DFS(G,v)
init() {
For each u ∈ G u.visited = false
For each u ∈ G DFS(G, u)
}
```

Program:-

```
graph = {
  '5': ['3','7'],
  '3': ['2', '4'],
  '7': [],
  '2': [],
  '4': ['8'],
  '8': []
}
visited = set()
def depthFirstSearch(visited, graph, node):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbour in graph[node]:
            depthFirstSearch(visited, graph, neighbour)
print("Depth First Search is:")
depthFirstSearch(visited, graph, '5')
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\DepthFirstSearch.py
Depth First Search is:
5 3 2 4 8 7
Process finished with exit code 0
```

EXPERIMENT:-3

Objective:- Write a program to implement Best First Search Algorithm.

Description:- An informed search, like BFS, uses an evaluation function to decide which among the various available nodes is the most promising (or 'BEST') before traversing to that node. BFS uses the concept of a Priority queue and heuristic search. To search the graph space, the BFS method uses two lists for tracking the traversal. An 'Open' list that keeps track of the current 'immediate' nodes available for traversal and a 'CLOSED' list that keeps track of the nodes already traversed.

Algorithm:-

- Create 2 empty lists: OPEN and CLOSED
- Start from the initial node (say N) and put it in the 'ordered' OPEN list
- Repeat the next steps until the GOAL node is reached
- If the OPEN list is empty, then EXIT the loop returning 'False'
- Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also, capture the information of the parent node
- If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path
- If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list
- Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by O(n*logn).

Pseudo Code:-

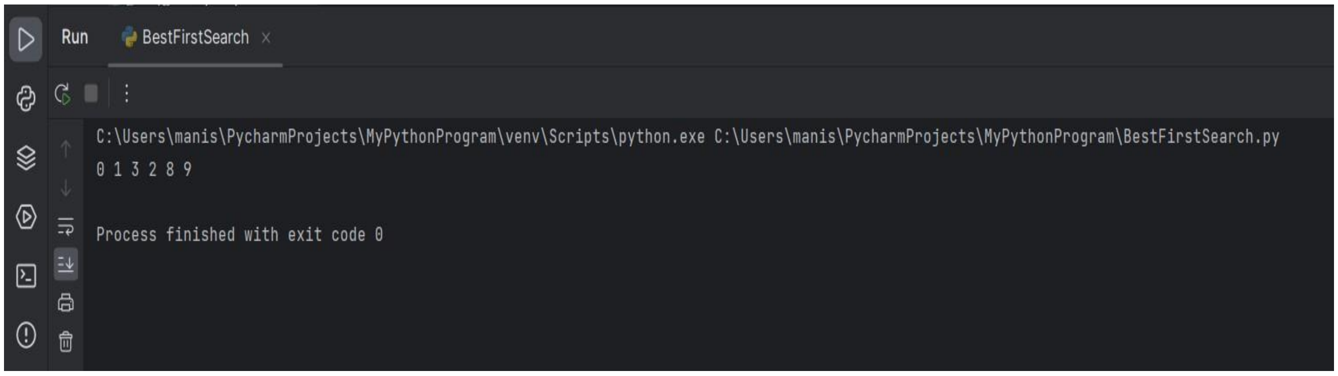
```
Best_First_Search(Graph g, Node start)
   1) Create an empty PriorityQueue
     PriorityQueue pq;
   2) Insert "start" in pq.
     pq.insert(start)
   3) Until PriorityQueue is empty
       u = PriorityQueue.DeleteMin
       If u is the goal
         Exit
       Else
         Foreach neighbor v of u
           If v "Unvisited"
              Mark v "Visited"
              pq.insert(v)
         Mark u "Examined"
End procedure
```

Program:-

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True
    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, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 9
best_first_search(source, target, v)
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\BestFirstSearch.py
0 1 3 2 8 9

Process finished with exit code 0
```

EXPERIMENT:-4

Objective:- Write a program to implement Mini – Max Algorithm for Game search.

Description:-

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state. In this algorithm two players play the game; one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Properties:

- Complete- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- Optimal- Min-Max algorithm is optimal if both opponents are playing optimally.
- Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(bm), where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

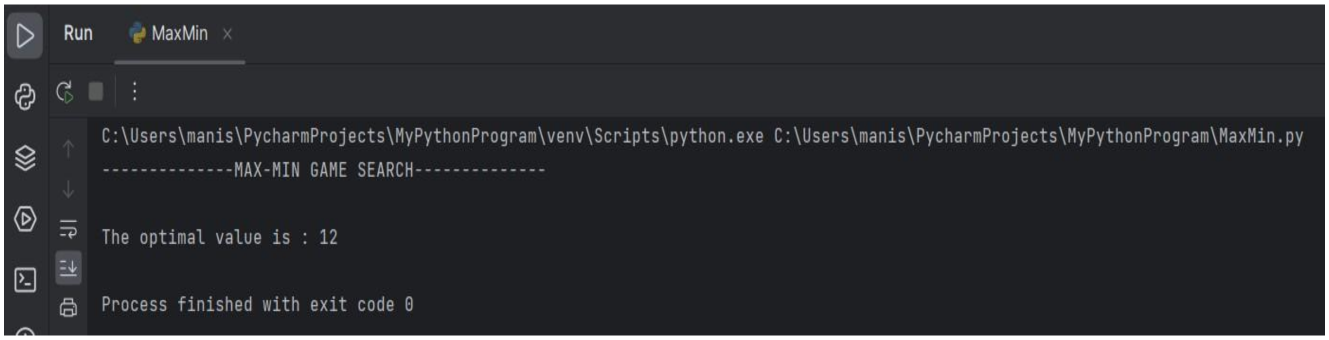
Pseudo Code:-

```
function minimax(node, depth, maximizingPlayer) is
    if depth ==0 or node is a terminal node then
        return static evaluation of node

    if MaximizingPlayer then
        maxEva= -infinity
        for each child of node do
            eva= minimax(child, depth-1, false)
            maxEva= max(maxEva,eva)
        return maxEva

    else
        minEva= +infinity
        for each child of node do
            eva= minimax(child, depth-1, true)
            minEva= min(minEva, eva)
        return minEva
```

Program:-

```
import math
def minimax(curDepth, nodeIndex,
            maxTurn, scores,
            targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,
                    False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,
                    True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,
                    True, scores, targetDepth))



# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
print("--------------MAX-MIN GAME SEARCH -------------\n")

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end="")
print(minimax(0, 0, True, scores, treeDepth))
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\MaxMin.py
-------------MAX-MIN GAME SEARCH-------------

The optimal value is : 12

Process finished with exit code 0
```

EXPERIMENT:-5

Objective:- Write a program to implement Alpha Beta Pruning Algorithm for Game Search.

Description:-

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to halfThis involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning.
- It is also called as Alpha- Beta Algorithm. Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prunes the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
    - ➢ Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximiser. The initial value of alpha is -∞.
    - ➢ Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.
- Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.
- The main condition which required for alpha-beta pruning is α>=β

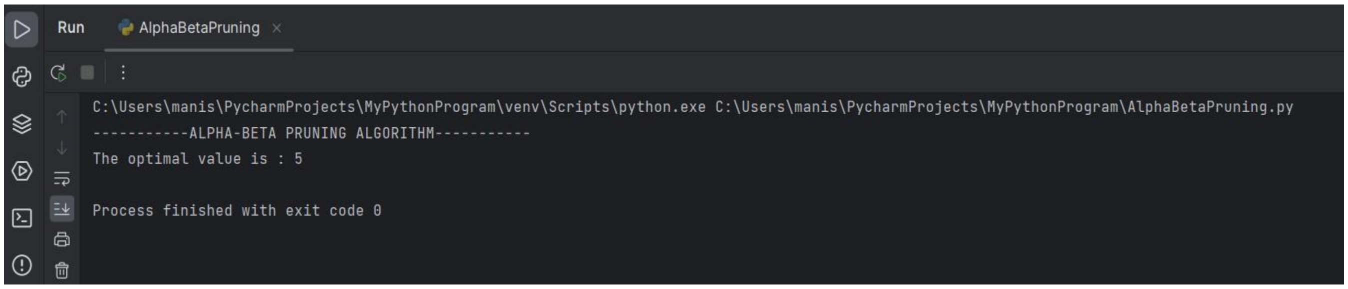
Pseudo Code:-

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
    if node is a leaf node :
        return value of the node
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
```

Program:-

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best
    # Driver Code
if__name__== "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("-----------ALPHA-BETA PRUNING ALGORITHM ----------")
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\AlphaBetaPruning.py
-----------ALPHA-BETA PRUNING ALGORITHM----------
The optimal value is : 5

Process finished with exit code 0
```

EXPERIMENT:-6

Objective:- Write a program to solve 8 – puzzle problem using Informed Search.

Description:- Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.

A* technique can be used to solve this problem. We use two lists namely 'open list' and 'closed list' the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after it's neighbouring nodes are discovered is put in the closed list and the neighbours are put in the open list this is how the nodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start(Initial) node. The next node chosen from the open list is based on its f score, the node with the least f score is picked up and explored.

f-score = h-score + g-score

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g- score (i.e. the number of nodes traversed from the start node to current node).

In our 8-Puzzle problem, we can define the h-score as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes. G-score will remain as the number of nodes traversed from start node to get to the current node.

Algorithm:-

- We first move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state.
- After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list.
- A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state.

Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path. This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.

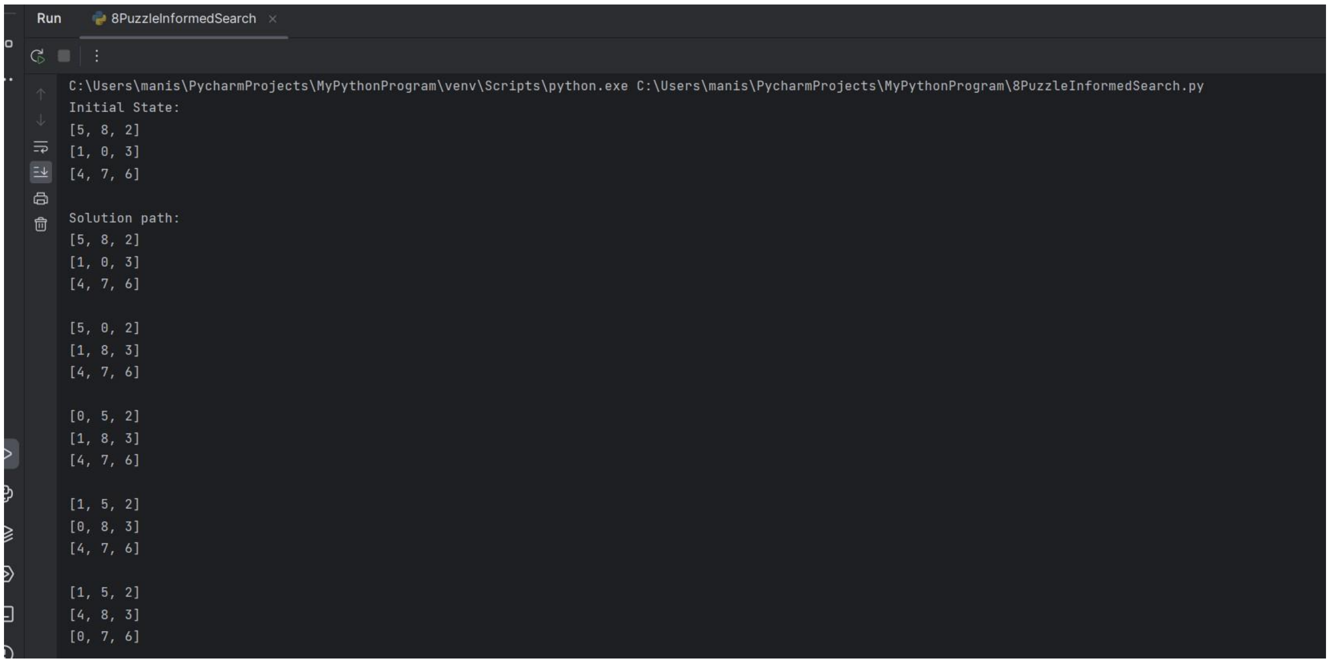
Pseudo Code:

```
function A-STAR-SEARCH(initialState, goalTest)
returns SUCCESS or FAILURE:
frontier = Heap.new(initialState)
explored = Set.new()
while not frontier.isEmpty();
state = frontier.deleteMin()
explored.add(state)
if goalTest(state):
return SUCCESS(state)
for neighbour in state.neughbour():
if neighbour not in frontier U explored:
frontier.insert(neighbour)
else if neighbour in frontier:
frontier.decreaseKey(neighbour)
return FAILURE
```

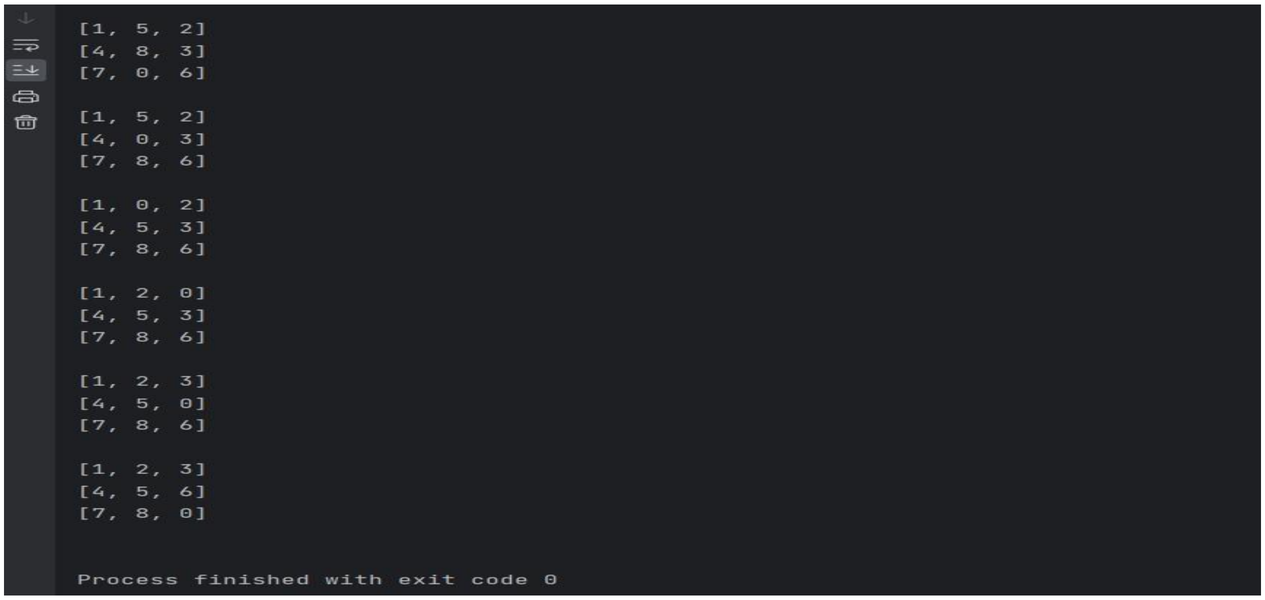
Program:-

```
import heapq

# Define the goal state and initial state of the 8-puzzle problem
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# Helper function to find the coordinates of a number in the puzzle
def find_number(puzzle, number):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == number:
                return i, j
# Helper function to calculate the Manhattan distance heuristic
def manhattan_distance(puzzle):
    distance = 0
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] != 0:
                goal_i, goal_j = find_number(goal_state, puzzle[i][j])
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

# Node class to represent a state in the search tree
class Node:
    def___init__(self, puzzle, g, h, parent):
        self.puzzle = puzzle
        self.g = g # cost to reach this state from the initial state
        self.h = h # heuristic value (Manhattan distance)
        self.parent = parent
    def f(self):
        return self.g + self.h

    def___lt__(self, other):
        return self.f() < other.f()
# A* search algorithm
def a_star(initial_state, goal_state):
    open_list = []
    closed_set = set()
    initial_node = Node(initial_state, 0, manhattan_distance(initial_state), None)
    heapq.heappush(open_list, initial_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.puzzle == goal_state:
            # Goal state is reached, reconstruct the path
            path = []
            while current_node:
                path.append(current_node.puzzle)
                current_node = current_node.parent
            return list(reversed(path))

        closed_set.add(tuple(map(tuple, current_node.puzzle)))

        for move in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
            new_puzzle = [list(row) for row in current_node.puzzle]
            i, j = find_number(new_puzzle, 0)
            new_i, new_j = i + move[0], j + move[1]

            if 0 <= new_i < 3 and 0 <= new_j < 3:
                new_puzzle[i][j], new_puzzle[new_i][new_j] = 
new_puzzle[new_i][new_j], new_puzzle[i][j]

                if tuple(map(tuple, new_puzzle)) not in closed_set:
                    new_node = Node(new_puzzle, current_node.g + 1, 
manhattan_distance(new_puzzle), current_node)
                    heapq.heappush(open_list, new_node)

    return None  # No solution found

# Example usage with a random initial state
import random

# Generate a random initial state
#initial_state = [[1, 6, 2], [3, 0, 8], [5, 4, 7]]
initial_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
for _ in range(50):  # Shuffle the initial state
    i, j = find_number(initial_state, 0)
    possible_moves = [(i, j-1), (i, j+1), (i-1, j), (i+1, j)]
    valid_moves = [(x, y) for x, y in possible_moves if 0 <= x < 3 and 0 <= y < 3]
    new_i, new_j = random.choice(valid_moves)
    initial_state[i][j], initial_state[new_i][new_j] = initial_state[new_i][new_j], 
initial_state[i][j]

# Find the solution path
solution_path = a_star(initial_state, goal_state)

if solution_path:
    print("Initial State:")
    for row in initial_state:
        print(row)
    print("\nSolution path:")
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\8PuzzleInformedSearch.py
Initial State:
[5, 8, 2]
[1, 0, 3]
[4, 7, 6]

Solution path:
[5, 8, 2]
[1, 0, 3]
[4, 7, 6]

[5, 0, 2]
[1, 8, 3]
[4, 7, 6]

[0, 5, 2]
[1, 8, 3]
[4, 7, 6]

[1, 5, 2]
[0, 8, 3]
[4, 7, 6]

[1, 5, 2]
[4, 8, 3]
[0, 7, 6]
```



```
[1, 5, 2]
[4, 8, 3]
[7, 0, 6]

[1, 5, 2]
[4, 0, 3]
[7, 8, 6]

[1, 0, 2]
[4, 5, 3]
[7, 8, 6]

[1, 2, 0]
[4, 5, 3]
[7, 8, 6]

[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Process finished with exit code 0
```

EXPERIMENT:-7

Objective:- Write a program to solve 8 – puzzle problem using Uninformed Search Algorithm.

Description:- Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.

Branch and Bound technique for solving this problem.

The search for an answer node can often be speeded by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS-like search.

There are basically three types of nodes involved in Branch and Bound:

- Live node is a node that has been generated but whose children have not yet been generated.
- E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Cost function:

- Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as
  $C(X) = g(X) + h(X)$ where
- $g(X)$ = cost of reaching the current node from the root
- $h(X)$ = cost of reaching an answer node from X. The ideal Cost function for an 8-puzzle Algorithm
- $c(x) = f(x) + h(x)$ where
- $f(x)$ is the length of the path from root to x (the number of moves so far) and $h(x)$ is the number of non-blank tiles not in their goal position (the number of mis- -placed tiles). There are at least $h(x)$ moves to transform state x to a goal state An algorithm is available for getting an approximation of $h(x)$ which is an unknown value.

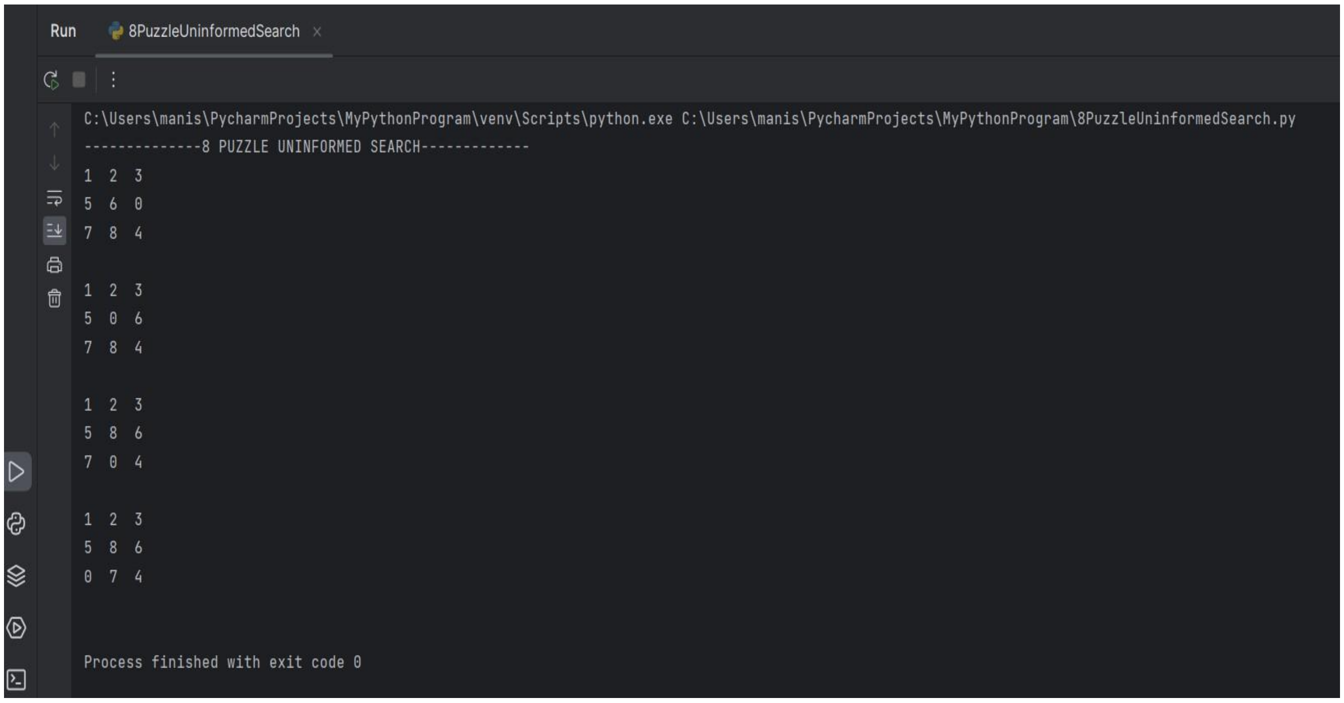
Pseudo Code:-

```
struct list_node
{
  list_node *next;
  // Helps in tracing path when answer is found
  list_node *parent;
  float cost;
}
algorithm LCSearch(list_node *t)
{
  // Search t for an answer node
  // Input: Root node of tree t
  // Output: Path from answer node to root
  if (*t is an answer node)
  {
     print(*t);
     return;
  }
  E = t; // E-node
  Initialize the list of live nodes to be empty;
  while (true)
  {
    for each child x of E
    {
       if x is an answer node
{
         print the path from x to t;
         return;
       }
       Add (x); // Add x to list of live nodes;
       x->parent = E; // Pointer for path to root
    }

    if there are no more live nodes
    {
      print ("No answer node");
      return;
    }

    // Find a live node with least estimated cost
    E = Least();
    // The found node is deleted from the list of
    // live nodes
  }
}
```

Program:-

```
import copy
from heapq import heappush, heappop
n = 3
# bottom, left, top, right
row = [1, 0, -1, 0]
col = [0, -1, 0, 1]

class priorityQueue:
    def__init_(self):
        self.heap = []
    # Inserts a new key 'k'
    def push(self, k):
        heappush(self.heap, k)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure
class node:
    def__init_(self, parent, mat, empty_tile_pos,
                cost, level):
        self.parent = parent
        self.mat = mat
        self.empty_tile_pos = empty_tile_pos
        self.cost = cost
        self.level = level
    def__lt_(self, nxt):
        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                    (mat[i][j] != final[i][j])):
                count += 1
    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]
    cost = calculateCost(new_mat, final)
    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node

def printMatrix(mat):
    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end=" ")
        print()

def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):
    pq = priorityQueue()
    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)
    pq.push(root)
    while not pq.empty():
        minimum = pq.pop()
        if minimum.cost == 0:
            printPath(minimum)
            return
        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]
            if isSafe(new_tile_pos[0], new_tile_pos[1]):
                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final, )

                pq.push(child)

initial = [[1, 2, 3],
           [5, 6, 0],
           [7, 8, 4]]

final = [[1, 2, 3],
         [5, 8, 6],
         [0, 7, 4]]

print("--------------8 PUZZLE UNINFORMED SEARCH------------ ")
empty_tile_pos = [1, 2]

solve(initial, empty_tile_pos, final)
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\8PuzzleUninformedSearch.py
-------------8 PUZZLE UNINFORMED SEARCH------------
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4


Process finished with exit code 0
```

EXPERIMENT:-8

Objective:- Write a program to implement A* Algorithm.

Description:- It is a searching algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. It searches for shorter paths first, thus making it an optimal and complete algorithm. Another aspect that makes A* so powerful is the use of weighted graphs in its implementation.

Algorithm:-

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function buthave not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.The algorithm is as follows-

Step-01:

- Define a list OPEN.
- Initially, OPEN consists solely of a single node, the start node S.

Step-02:

If the list is empty, return failure and exit.

Step-03:

- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
- If node n is a goal state, return success and exit.

Step-04:

Expand node n.

Step-05:

- If any successor to n is the goal node, return success and the solution by tracing thepath from goal node to S.
- Otherwise, go to Step-06.

Step-06:

For each successor node,

- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

Step-07:

Go back to Step-02.

Pseudo Code:-

```
Initial condition - we create two lists - Open List and Closed List.Now, the following steps need to be
implemented -
  • The open list must be initialized.
  • Put the starting node on the open list (leave its f at zero). Initialize the closed list.
  • Follow the steps until the open list is non-empty:
      ➢ Find the node with the least f on the open list and name it "q".
      ➢ Remove Q from the open list.
      ➢ Produce q's eight descendants and set q as their parent.
      ➢ For every descendant:
      i)  If finding a successor is the goal, cease looking
      ii) Else, calculate g and h for the successor.
          successor.g = q.g + the calculated distance between the successor and the q.
```

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

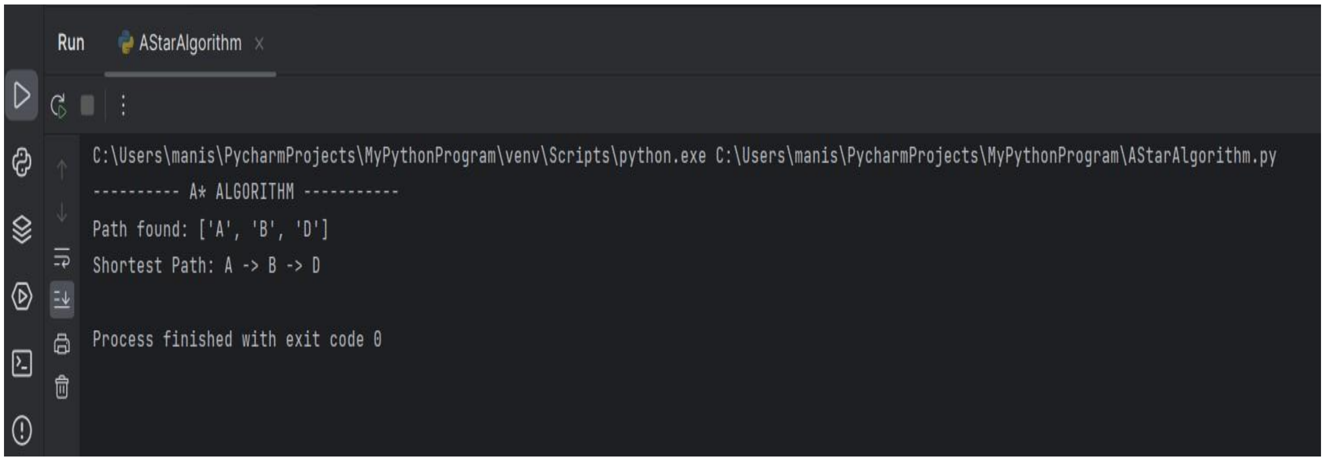
iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

- Push Q into the closed list and end the while loop.

Program:-

```
from collections import deque

class Graph:

    def__init_(self, adjac_lis):

        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):

        return self.adjac_lis[v]

    def h(self, n):

        H = {

            'A': 1,

            'B': 1,

            'C': 1,

            'D': 1

        }
        return H[n]

    def a_star_algorithm(self, start, stop):

        open_lst = set([start])

        closed_lst = set()

        poo = {}

        poo[start] = 0

        par = {}

        par[start] = start

        while len(open_lst) > 0:

            n = None

            for v in open_lst:

                if (n is None or poo[v] + self.h(v) < poo[n] + self.h(n)):

                    n = v

            if n is None:

                print('Path does not exist!')

                return None

            if n == stop:

                reconst_path = []

                while par[n] != n:

                    reconst_path.append(n)

                    n = par[n]
                reconst_path.append(start)

                reconst_path.reverse()

                print('Path found:', reconst_path)

                return reconst_path

            for (m, weight) in self.get_neighbors(n):

                if m not in open_lst and m not in closed_lst:

                    open_lst.add(m)

                    par[m] = n

                    poo[m] = poo[n] + weight

                else:

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

                        poo[m] = poo[n] + weight

                        par[m] = n

                        if m in closed_lst:

                            closed_lst.remove(m)

                            open_lst.add(m)

            open_lst.remove(n)

            closed_lst.add(n)

        print('Path does not exist!')

        return None

adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
print("---------- A* ALGORITHM----------- ")
graph1 = Graph(adjac_lis)
path = graph1.a_star_algorithm('A', 'D')
if path:
    print("Shortest Path:", ' -> '.join(path))
```

Output:-



```
C:\Users\manis\PycharmProjects\MyPythonProgram\venv\Scripts\python.exe C:\Users\manis\PycharmProjects\MyPythonProgram\AStarAlgorithm.py
---------- A* ALGORITHM -----------
Path found: ['A', 'B', 'D']
Shortest Path: A -> B -> D

Process finished with exit code 0
```

EXPERIMENT:-9

Objective:- Write a program to construct a Bayesian network from given data.

Description:- A Bayesian network is a directed acyclic graph in which each edge corresponds to a conditional dependency, and each node corresponds to a unique random variable. Bayesian network consists of two major parts:

- ○ a directed acyclic graph and
- ○ a set of conditional probability distributions
- ▪ The directed acyclic graph is a set of random variables represented by nodes.
- ▪ The conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

Directed Acyclic Graph:

A Directed Acyclic Graph is used to represent a Bayesian Network and like any other statistical graph, a DAG contains a set of nodes and links, where the links denote the relationship between the nodes. The nodes here represent random variables and the edges define the relationship between these variables.
A DAG models the uncertainty of an event occurring based on the Conditional Probability Distribution (CDP) of each random variable. A Conditional Probability Table (CPT) is used to represent the CPD of each variable in the network.

Using Bayesian Networks to solve the famous Monty Hall Problem:

The Monty Hall problem named after the host of the TV series, 'Let's Make A Deal', is a paradoxical probability puzzle that has been confusing people for over a decade. So this is how it works. The game involves three doors, given that behind one of these doors is a car and the remaining two have goats behind them. So you start by picking a random door,
say#2. On the other hand, the host knows where the car is hidden and he opens another door,
say #1 (behind which there is a goat). Here's the catch, you're now given a choice, the host will ask you if you want to pick door #3 instead of your first choice i.e. #2.

Algorithm:-

- The first step is to build a Directed Acyclic Graph.
- The graph has three nodes, each representing the door chosen by:
    - The door selected by the Guest
    - The door containing the prize (car)
    - The door Monty chooses to open

Let’s understand the dependencies here, the door selected by the guest and the door containing the car are completely random processes. However, the door Monty chooses to open is dependent on both the doors; the door selected by the guest, and the door the prize is behind. Monty has to choose in such a way that the door does not contain the prize and it cannot be the one chosen by the guest.

Program:-

```
Initial
```