| Ex.No:	
Date:	

AIM:

DESCRIPTION:

In DFS, we continue to traverse downwards through linked nodes until we reach the end, then retrace our steps to check which connected nodes we haven't visited and repeat the process. In a depth-first search, we dive deep into the graph and then backtrack when we reach the bottom.

As we discussed above, the DFS is like solving a maze. We have to continue to walk along the path until we reach the end.

Let's have a look at the below example. We start with node 8, then go down to 3, then down to 1, and now we have hit the bottom, so start backtracking. Now at 3, we have one unvisited node, so go to 6, go down to 4, now hit the bottom, so backtrack again and reach node 6. Now we have one more unvisited node, i.e., 7, so go down to 7, now hit the bottom, so directly backtrack to node 8, now repeat the process.
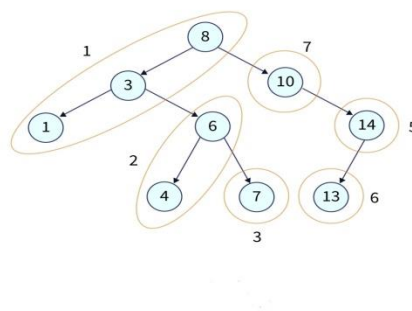
The depth-first algorithm chooses one path and continues it until it reaches the end.

We created a graph with five nodes in the code. Our source node is 0. We have a visited array that stores all the nodes that have been visited and a component array that stores the answer. Then we call the dfs

function, passing it to the source node, visited array, component array, and graph. When we call the dfs function with any node, we first visit that node and add it to the answer array. Then we visit all of the adjacent node nodes of the given node.

ALGORITHM :

To implement DFS, we use the Stack data structure to keep track of the visited nodes. We begin with the root as the first element in the stack, then pop from it and add all the related nodes of the popped node to the stack. We repeated the process until the stack became empty.

Every time we reach a new node, we will take the following steps:

1. We add the node to the top of the stack.

2. Marked it as visited.

3. We check if this node has any adjacent nodes:

    1. If it has adjacent nodes, then we ensure that they have not been visited already, and then visited it.

    2. We removed it from the stack if it had no adjacent nodes.

With every node added to the stack, we repeat the above steps or recursively visit each node until we reach the dead end.

PROGRAM:

```python
# Python program to print DFS traversal for complete graph
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)
```

```python
        # A function used by DFS
        def DFSUtil(self, v, visited):

                # Mark the current node as visited and print it
                visited[v]= True
                print (v)

                # Recur for all the vertices adjacent to
                # this vertex
                for i in self.graph[v]:
                        if visited[i] == False:
                                self.DFSUtil(i, visited)


        # The function to do DFS traversal. It uses
        # recursive DFSUtil()
        def DFS(self):
                V = len(self.graph) #total vertices

                # Mark all the vertices as not visited
                visited =[False]*(V)

                # Call the recursive helper function to print
                # DFS traversal starting from all vertices one
                # by one
                for i in range(V):
                        if visited[i] == False:
                                self.DFSUtil(i, visited)


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Depth First Traversal")
g.DFS()
```

OUTPUT:


```
IDLE Shell 3.11.3                                                                                    –  ☐  ×
File  Edit  Shell  Debug  Options  Window  Help
     Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
     Type "help", "copyright", "credits" or "license()" for more information.
>>>
     ================================ RESTART: C:/Users/dhars/OneDrive/Desktop/AI_LAB/DFS.py ================================
     Following is Depth First Traversal
     0
     1
     2
     3
>>>
                                                                                                Ln: 10  Col: 0
```

| PAAVAI ENGINEERING COLLEGE		
(Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited.
- Not visited.
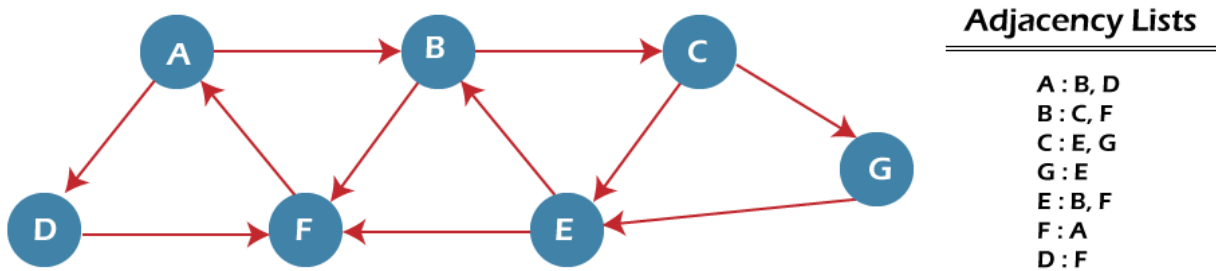
Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

ALGORITHM:

Step 1: Consider the graph you want to navigate.

Step 2: Select any vertex in your graph, Say v1, from which you want to traverse the graph.

Step 3: Examine any two data structure for traversing the graph.

Visited array(size of the graph)

Queue data structure

Step 4: Starting from the vertex, you will add to the visited array, and afterward, you will v1's adjacent vertices to the queue data structure.

Step 5: Now, using the FIFO concept, you must remove the element from the queue, put it into the visited array, and then return to the queue to add the adjacent vertices of the removed element.

Step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.

PROGRAM:

```python
# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)
# traverses vertices reachable from s. from collections import defaultdict


# This class represents a directed graph

# using adjacency list representation



class Graph:


        # Constructor
        def __init__(self):


                # default dictionary to store graph
                self.graph = defaultdict(list)


        # function to add an edge to graph
        def addEdge(self, u, v):
                self.graph[u].append(v)



        # Function to print a BFS of graph
        def BFS(self, s):

                # Mark all the vertices as not visited

                visited = [False] * (max(self.graph) + 1)

                # Create a queue for BFS

                queue = []


                # Mark the source node as

                # visited and enqueue it

                queue.append(s)
                visited[s] = True
```

```python
        while queue:

            s = queue.pop(0)
            print(s, end=" ")

            # Get all adjacent vertices of the

            # dequeued vertex s. If a adjacent

            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code


# Create a graph given in # the above diagram
g = Graph()
g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)


print("Following is Breadth First Traversal"
" (starting from vertex 2)")
g.BFS(2)
```

OUTPUT:

```
======
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
>>>
```

PAAVAI ENGINEERING COLLEGE		
(Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

      The best First Search algorithm in artificial intelligence is used for for finding the shortest path from a given starting node to a goal node in a graph. The algorithm works by expanding the nodes of the graph in order of increasing the distance from the starting node until the goal node is reached.
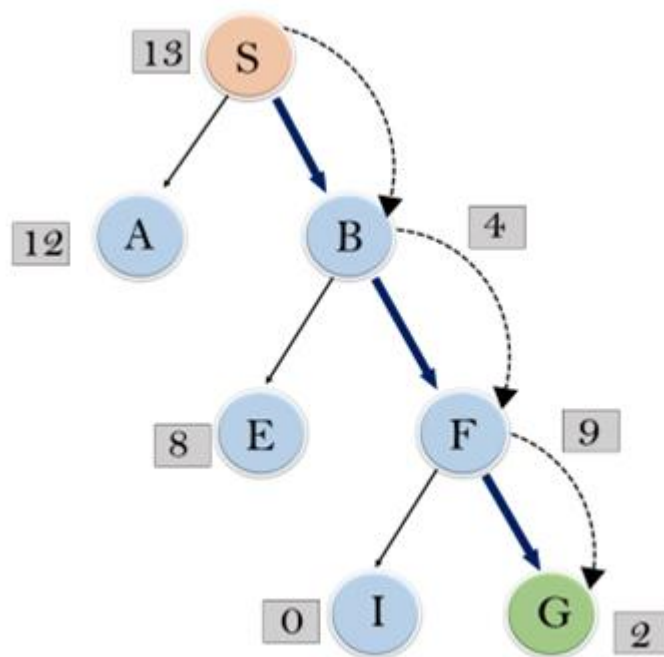
Initialization: Open [A, B], Closed [S]

Step 1: Open [A], Closed [S, B]

Step 2: Open [E, F, A], Closed [S, B]

       Open [E, A], Closed [S, B, F]

Step 3: Open [I, G, E, A], Closed [S, B, F]

       Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

ALGORITHM:

Step 1: Start the Program.

Step 2: Create 2 empty lists: OPEN and CLOSED.

Step 3: Start from the initial node (say N) and put it in the 'ordered' OPEN list.

Step 4: Repeat the next steps until the GOAL node is reached. If the OPEN list is empty, then EXIT the loop returning 'False'.

Step 5: Stop the Program.

PROGRAM:

```
from queue import PriorityQueue
v = 14
graph =[[] for i in range(v)]


# Function For Implementing Best First Search

# Gives output path having lowest cost



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:
```

```python
                u = pq.get()[1]

                # Displaying the path having lowest cost

                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()


# Function for adding edges to graph

def addedge(x, y, cost):

        graph[x].append((y, cost))
        graph[y].append((x, cost))


# The nodes shown in above example(by alphabets) are # implemented using integers addedge(x,y,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)
```
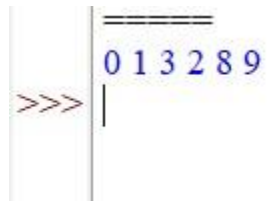
OUTPUT:

```
=====
013289
>>>
```

| PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

You are given a list of n cities with the distance between any two cities. Now, you have to start with your office and to visit all the cities only once each and return to your office. What is the shortest path can you take? This problem is called the Traveling Salesman Problem (TSP).
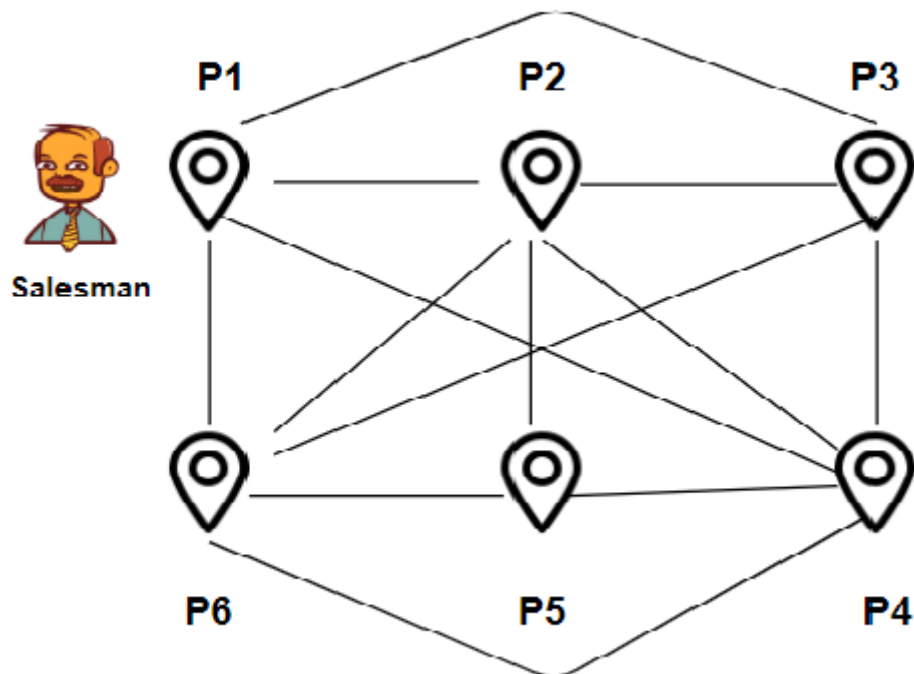
ALGORITHM

Step 1: Start the Program.

Step 2: Consider city 1 as the starting and ending point.

Step 3: Generate all (n-1)! Permutations of cities.

Step 4: Calculate the cost of every permutation and keep track of the minimum cost permutation.

Step 5: Return the permutation with minimum cost.

Step 6: Stop the Program.

PROGRAM:

```python
# Python3 program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4


# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

        # store all vertex apart from source vertex
        vertex = []
        for i in range(V):
                if i != s:

                        vertex.append(i)


        # store minimum weight Hamiltonian Cycle
        min_path = maxsize
```

```python
        next_permutation=permutations(vertex)
        for i in next_permutation:

                # store current Path weight(cost)
                current_pathweight = 0

                # compute current path weight
                k = s
                for j in i:
                        current_pathweight += graph[k][j]
                        k = j
                current_pathweight += graph[k][s]

                # update minimum
                min_path = min(min_path, current_pathweight)

        return min_path


# Driver Code
if __name__ == "__main__":

        # matrix representation of graph
        graph = [[0, 10, 15, 20], [10, 0, 35, 25],[15, 35, 0, 30], [20, 25, 30, 0]]
        s = 0
        print(travellingSalesmanProblem(graph, s))
```

OUTPUT:


```
Python 3.11.3 (tags/v3.11.3:f3909b8, Apr  4 2023, 23:49:59) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/dhars/OneDrive/Desktop/AI_LAB/Travelling_Salesman_Problem.py
80
>>>
```

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	**MAX. MARKS**	**MARKS AWARDED**
Preparation & Conduction	**10**	
Observation & Results	**20**	
Record Completion	**05**	
Viva Voce	**05**	
TOTAL	**40**	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How canyou get exactly 2 gallons of water in the 4-gallon jug?

Let X represent the content of the water in 4-gallon jug.

Let Y represent the content of the water in 3-gallon jug.

Write a program in python to define a set of operators (Rules) that will take us from

one state to another:

Start from initial state (X=0, Y=0)

Reach any of the Goal states

 (X=2, Y=0)

 (X=2, Y=1)

 (X=2, Y=2)

 (X=2, Y=3)

Find the minimum number of steps to reach any the above mentioned goal states.

Production Rules:-
R1: (x,y) --> (4,y) if x < 4

R2: (x,y) --> (x,3) if y < 3
R3: (x,y) --> (x-d,y) if x > 0
R4: (x,y) --> (x,y-d) if y > 0
R5: (x,y) --> (0,y) if x > 0
R6: (x,y) --> (x,0) if y > 0
R7: (x,y) --> (4,y-(4-x))
    if x+y >= 4 and y > 0
R8: (x,y) --> (x-(3-y),y)
   if x+y >=3 and x>0
R9: (x,y) --> (x+y,0)
   if x+y=<4 and y>0
R10: (x,y) --> (0,x+y)
   if x+y =< 3 and x>0

ALGORITHM:

Step 1: Start the Program.

Step 2: Empty the three gallon jug

Step 3: Now empty the remaining two gallons from the two gallon jug.

Step 4: Next refill the two gallon jug and empty one gallon from it into the three gallon jug. This gives four gallons.

Step 5: Stop the Program.

PROGRAM:

```
# This function is used to initialize the

# dictionary elements with a default value.

from collections import defaultdict

# jug1 and jug2 contain the value

# for max capacity in respective jugs

# and aim is the amount of water to be measured.

jug1, jug2, aim = 4, 3, 2
```

```python
# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)


# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

        # Checks for our goal and
        # returns true if achieved.
        if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
                print(amt1, amt2)
                return True


        # Checks if we have already visited the
        # combination or not. If not, then it proceeds further.
        if visited[(amt1, amt2)] == False:
                print(amt1, amt2)

                # Changes the boolean value of
                # the combination as it is visited.
                visited[(amt1, amt2)] = True

                # Check for all the 6 possibilities and
                # see if a solution is found in any one of them.
                return (waterJugSolver(0, amt2) or
```

```python
                    waterJugSolver(amt1, 0) or

                    waterJugSolver(jug1, amt2) or

                    waterJugSolver(amt1, jug2) or

                    waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

                    amt2 - min(amt2, (jug1-amt1))) or

                    waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

                    amt2 + min(amt1, (jug2-amt2))))


        # Return False if the combination is
        # already visited to avoid repetition otherwise
        # recursion will enter an infinite loop.
        else:

                return False


print("Steps: ")


# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```
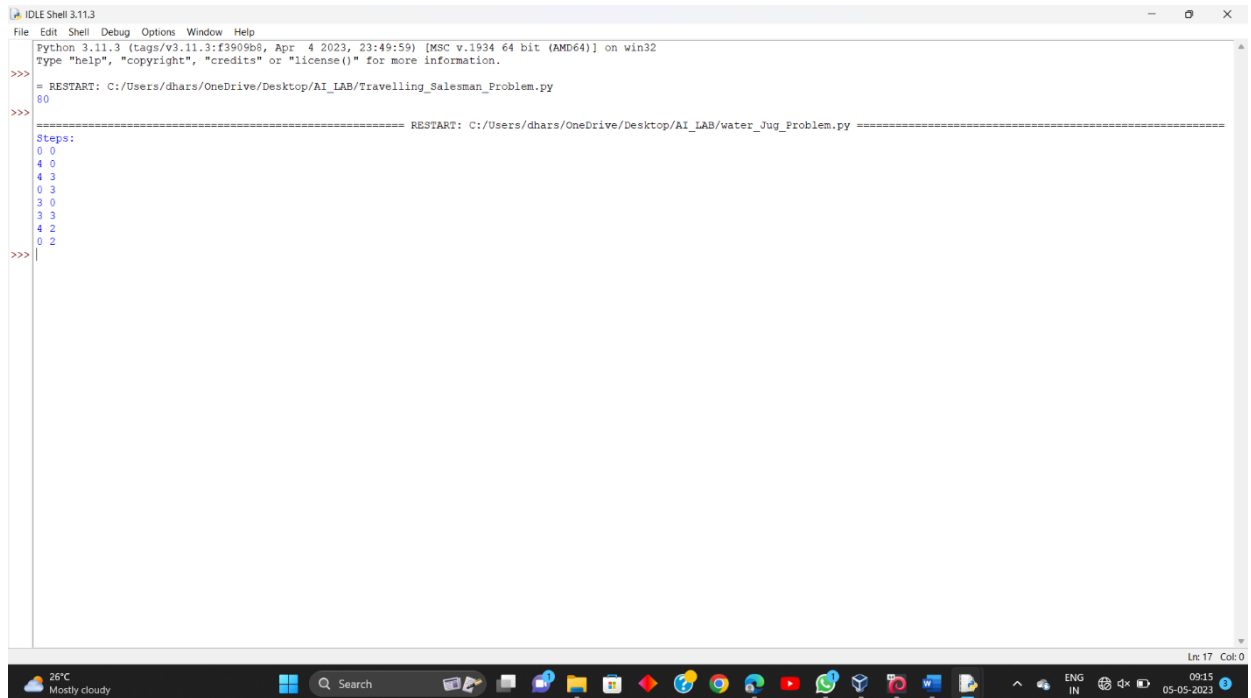
OUTPUT:

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	**MAX. MARKS**	**MARKS AWARDED**
Preparation & Conduction	**10**	
Observation & Results	**20**	
Record Completion	**05**	
Viva Voce	**05**	
TOTAL	**40**	

**RESULT:**

Ex.No:	
Date:	

AIM:

DESCRIPTION:

The Tower of Hanoi (also called "The problem of Benares Temple" or "Tower of Brahma" or Lucas' Tower and sometimes pluralized as Towers, or simply pyramid puzzle) is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod.

This object of this famous puzzle is to move N disks from the left peg to the right peg using the center peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk. The following diagram depicts the starting setup for N=3 disks.

Let us we have three disks stacked on a peg
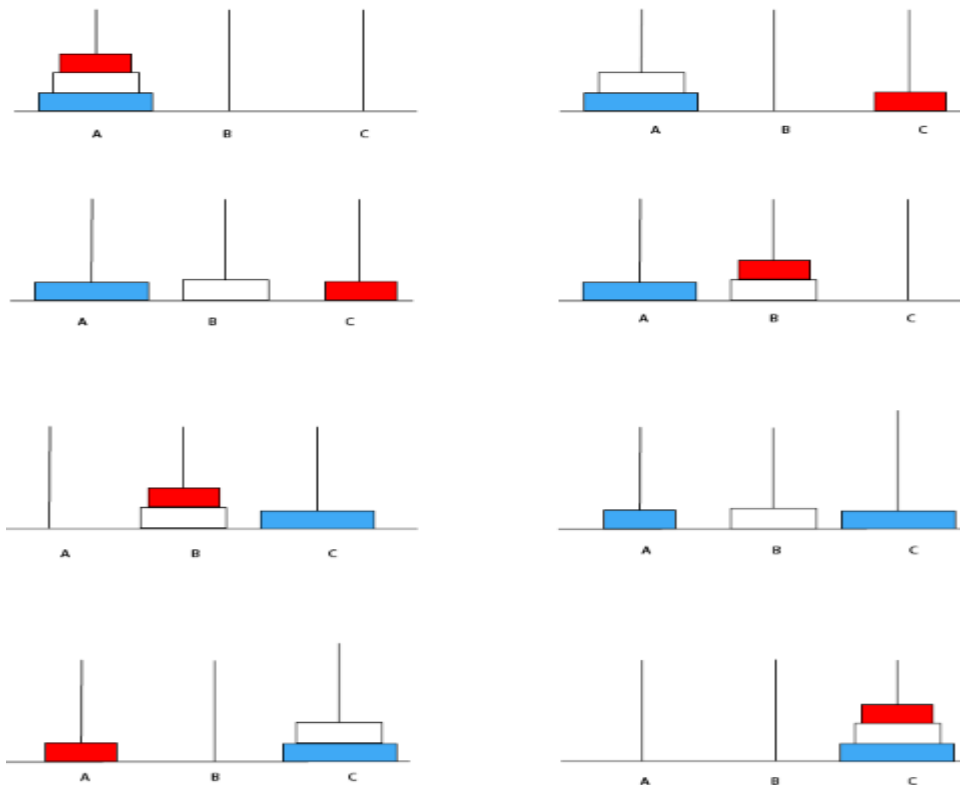
1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.
3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

ALGORITHM:

Step 1: Start

Step 2: Let the three towers be the source, dest, aux.

Step 3: Read the number of disks, n from the user.

Step 4: Move n-1 disks from source to aux.

Step 5: Move nth disk from source to dest.

Step 6: Move n-1 disks from aux to dest.

Step 7: Repeat Steps 3 to 5, by decrementing n by 1.

Step 8: Stop

PROGRAM:

```python
# Recursive Python function to solve tower of hanoi

def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:

        return

    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)
```

```python
        TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

# Driver code
N = 3


# A, C, B are the name of rods
TowerOfHanoi(N, 'A', 'C', 'B')
```

OUTPUT:

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

| PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).
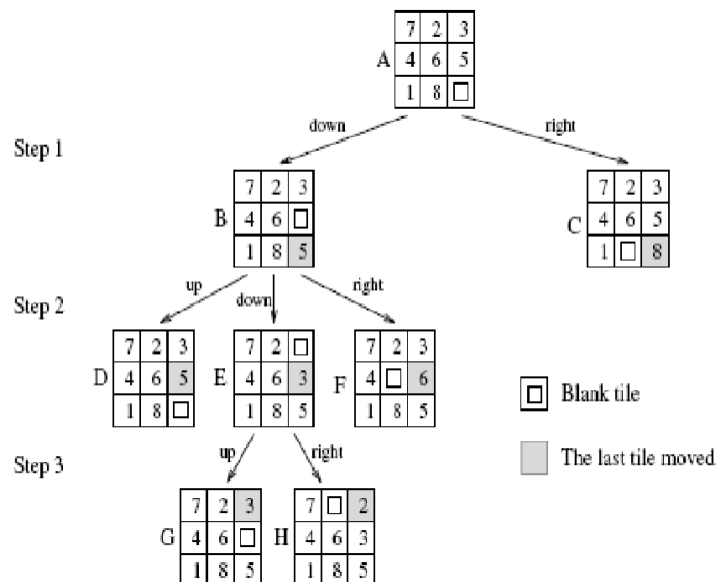
ALGORITHM:

Step 1: Start

Step 2: We first move the empty space in all the possible directions in the start state and calculate the f-score for each state.

Step 3: 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.

Step 4: This is called expanding the current state. This process continues until the goal state occurs as the current state.

Step 5: 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.

Step 6: This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.

Step 7: Stop

PROGRAM:

```python
# Python3 program to print the path from root

# node to destination node for N*N-1 puzzle

# algorithm using Branch and Bound

# The solution assumes that instance of

# puzzle is solvable


# Importing copy for deepcopy function

import copy


# Importing the heap functions from python

# library for Priority Queue

from heapq import heappush, heappop


# This variable can be changed to change
```

```python
# the program from 8 puzzle(n=3) to 15

# puzzle(n=4) to 24 puzzle(n=5)...

n = 3


# bottom, left, top, right

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]


# A class for Priority Queue

class priorityQueue:


    # Constructor to initialize a

    # Priority Queue

    def _init_(self):

        self.heap = []


    # Inserts a new key 'k'

    def push(self, k):

        heappush(self.heap, k)


    # Method to remove minimum element

    # from Priority Queue

    def pop(self):

        return heappop(self.heap)


    # Method to know if the Queue is empty

    def empty(self):

        if not self.heap:

                return True

        else:
```

```python
            return False

# Node structure
class node:

    def _init_(self, parent, mat, empty_tile_pos, cost, level):

        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

        # Stores the matrix
        self.mat = mat

        # Stores the position at which the
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Stores the number of misplaced tiles
        self.cost = cost

        # Stores the number of moves so far
        self.level = level

    # This method is defined so that the
    # priority queue is formed based on
    # the cost variable of the objects
    def _lt_(self, nxt):
        return self.cost < nxt.cost
```

```python
# Function to calculate the number of
# misplaced tiles ie. number of non-blank
# tiles not in their goal position
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:

    # Copy data from parent matrix to current matrix
    new_mat = copy.deepcopy(mat)

    # Move tile by 1 position
    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]

    # Set number of misplaced tiles
    cost = calculateCost(new_mat, final)
```

```python
        new_node = node(parent, new_mat, new_empty_tile_pos, cost, level)
    return new_node
# Function to print the N x N matrix
def printMatrix(mat):

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

    print()

# Function to check if (x, y) is a valid
# matrix coordinate
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)
    print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state.
```

```python
def solve(initial, empty_tile_pos, final):
    # Create a priority queue to store live
    # nodes of search tree
    pq = priorityQueue()

    # Create the root node
    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)

    # Add root to list of live nodes
    pq.push(root)

    # Finds a live node with least cost,
    # add its children to list of live
    # nodes and finally deletes it from
    # the list.
    while not pq.empty():

        # Find a live node with least estimated
        # cost and delete it from the list of
        # live nodes
        minimum = pq.pop()

        # If minimum is the answer node
        if minimum.cost == 0:

            # Print the path from root to
            # destination;
            printPath(minimum)
```

```python
        return

    # Generate all possible children
    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]):

            # Create a child node
            child = newNode(minimum.mat, minimum.empty_tile_pos, new_tile_pos,
                minimum.level + 1, minimum, final,)

            # Add child to list of live nodes
            pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
        [ 5, 6, 0 ],
        [ 7, 8, 4 ] ]

# Solvable Final configuration
# Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
[ 5, 8, 6 ],
[ 0, 7, 4 ] ]
```
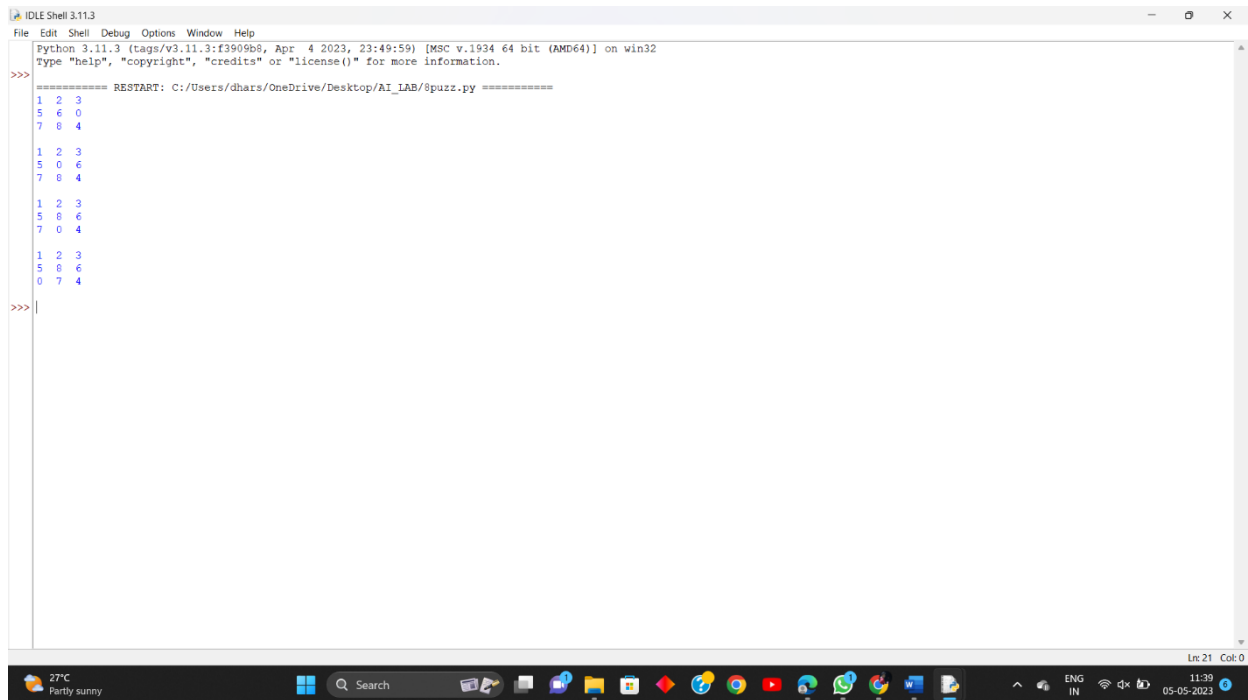
```
# Blank tile coordinates in
# initial configuration
empty_tile_pos = [ 1, 2 ]


# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)
```

OUTPUT:

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm
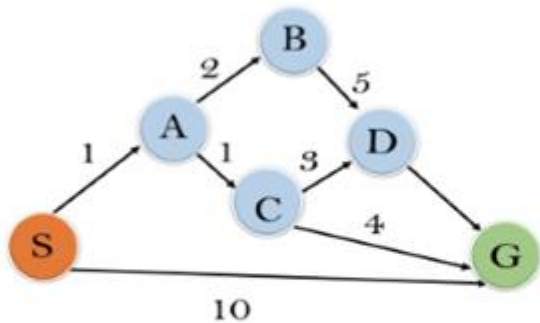
State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Initialization: {(S, 5)}

Step 1: {(S--> A, 4), (S-->G, 10)}

Step 2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Step 3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Step 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

ALGORITHM:

Step 1: Initialize the open list.

Step 2: Initialize the closed list put the starting node on the open list (you can leave its f at zero).

Step 3: while the open list is not empty

a) find the node with the least f on the open list, call it "q".

b) pop q off the open list.

c) generate q's 8 successors and set their parents to q.

d) for each successor.

     i)   if successor is the goal, stop search

     ii)   else, compute both g and h for successor successor.g = q.g + distance between successor and successor.h = distance from goal to successor (This can be done using many ways, we will discuss three heuristics-Manhattan, Diagonal and Euclidean Heuristics)successor.f = successor.g + successor.h

     iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

     iV) if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list end (for loop)

e) push q on the closed list end (while loop).

PROGRAM:

from collections import deque

class Graph:

   # example of adjacency list (or rather map)

```python
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }


def __init__(self, adjacency_list):
    self.adjacency_list = adjacency_list

def get_neighbors(self, v):
    return self.adjacency_list[v]


# heuristic function with equal values for all nodes
def h(self, n):
    H = {
        'A': 1,
        'B': 1,
        'C': 1,
        'D': 1
    }

    return H[n]


def a_star_algorithm(self, start_node, stop_node):
    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
```

```python
# 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):
```

```python
            # 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


adjacency_list = {

    'A': [('B', 1), ('C', 3), ('D', 7)],

    'B': [('D', 5)],

    'C': [('D', 12)]

}

graph1 = Graph(adjacency_list) graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

Path found: ['A', 'B', 'D']
['A', 'B', 'D']

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

The AO* algorithm is a knowledge-based search technique, meaning the start state and the goal state is already defined , and the best path is found using heuristics. The time complexity of the algorithm is significantly reduced due to the informed search technique.
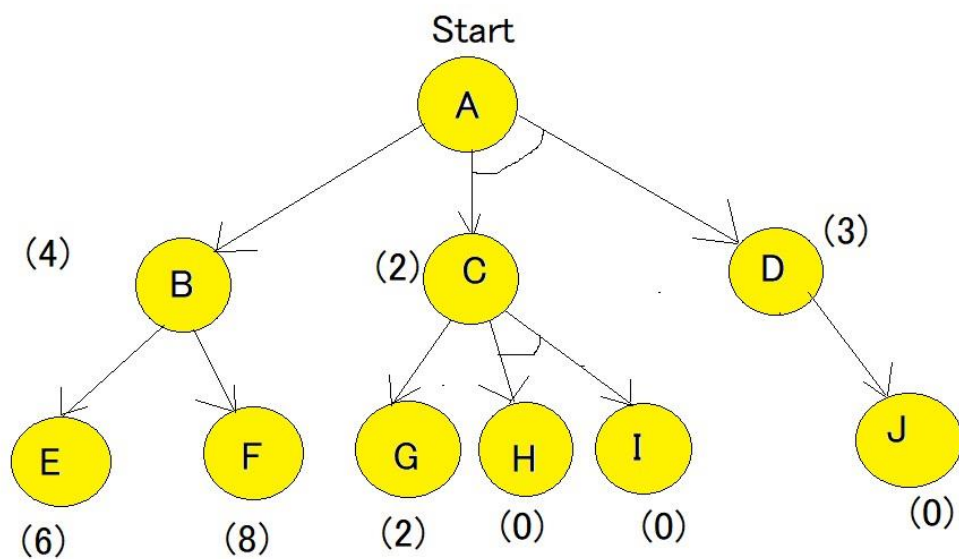
Step-1:
Starting from node A, we first calculate the best path.
f(A-B) = g(B) + h(B) = 1+4= 5 , where 1 is the default cost value of travelling from A to B and 4 is the estimated cost from B to Goal state.
f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7 , here we are calculating the path cost as both C and D because they have the AND-Arc. The default cost value of travelling from A-C is 1, and from A-D is 1, but the heuristic value given for C and D are 2 and 3 respectively hence making the cost as 7.

Step-2:

Using the same formula as step-1, the path is now calculated from the B node,

f(B-E) = 1 + 6 = 7.

f(B-F) = 1 + 8 = 9

Hence, the B-E path has lesser cost. Now the heuristics have to be updated since there is a difference between actual and heuristic value of B. The minimum cost path is chosen and is updated as the heuristic , in our case the value is 7. And because of change in heuristic of B there is also change in heuristic of A which is to be calculated again.

f(A-B) = g(B) + updated((h(B)) = 1+7=8

Step-3:

Comparing path of f(A-B) and f(A-C-D) it is seen that f(A-C-D) is smaller. Hence f(A-C-D) needs to be explored.

Now the current node becomes C node and the cost of the path is calculated,

f(C-G) = 1+2 = 3

f(C-H-I) = 1+0+1+0 = 2

f(C-H-I) is chosen as minimum cost path,also there is no change in heuristic since it matches the actual cost. Heuristic of path of H and I are 0 and hence they are solved, but Path A-D also needs to be calculated , since it has an AND-arc.

f(D-J) = 1+0 = 1, hence heuristic of D needs to be updated to 1. And finally the f(A-C-D) needs to be updated.

f(A-C-D) = g(C) + h(C) + g(D) + updated((h(D)) = 1+2+1+1 =5.

ALGORITHM:

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in CLOSE.

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit.

PROGRAM:

```python
# Cost to find the AND and OR path def Cost(H, condition, weight = 1):
        cost = {}

        if 'AND' in condition:

                AND_nodes = condition['AND']

                Path_A = ' AND '.join(AND_nodes)

                PathA = sum(H[node]+weight for node in AND_nodes)

                cost[Path_A] = PathA


        if 'OR' in condition:

                OR_nodes = condition['OR']

                Path_B =' OR '.join(OR_nodes)

                PathB = min(H[node]+weight for node in OR_nodes)
                cost[Path_B] = PathB
                return cost


# Update the cost def update_cost(H, Conditions, weight=1):
Main_nodes = list(Conditions.keys())
Main_nodes.reverse()
least_cost= {}
for key in Main_nodes:
    condition = Conditions[key]

     print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))

    c = Cost(H, condition, weight)

   H[key] = min(c.values())

   least_cost[key] = Cost(H, condition, weight)

        return least_cost

# Print the shortest path def shortest_path(Start,Updated_cost, H):
        Path = Start

        if Start in Updated_cost.keys():

Min_cost = min(Updated_cost[Start].values())
key = list(Updated_cost[Start].keys())
```

```python
            values = list(Updated_cost[Start].values())
                    Index = values.index(Min_cost)


                # FIND MINIMIMUM PATH KEY

                    Next = key[Index].split()

# ADD TO PATH FOR OR PATH
                    if len(Next) == 1:
                        Start =Next[0]

                        Path += '<--' +shortest_path(Start, Updated_cost, H)

                # ADD TO PATH FOR AND PATH

                    else:

                            Path +='<--('+key[Index]+') '


                            Start = Next[0]

                            Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '


                            Start = Next[-1]

                            Path += shortest_path(Start, Updated_cost, H) + ']'


        return Path



H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}


Conditions = {
'A': {'OR': ['B'], 'AND': ['C', 'D']},
'B': {'OR': ['E', 'F']},
'C': {'OR': ['G'], 'AND': ['H', 'I']},
'D': {'OR': ['J']}
}
# weight weight = 1 # Updated cost print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
```

```python
print('*'*75) print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

OUTPUT:

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
*************************************************************************

Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

RESULT:

| Ex.No:	
Date:	

AIM:

DESCRIPTION:

An expert system is a computer program that uses artificial intelligence (AI) technologies to simulate the judgment and behavior of a human or an organization that has expertise and experience in a particular field. Expert systems are usually intended to complement, not replace, human experts.

Modern expert knowledge systems use machine learning and artificial intelligence to simulate the behavior or judgment of domain experts. These systems can improve their performance over time as they gain more experience, just as humans do.

Expert systems accumulate experience and facts in a knowledge base and integrate them with an inference or rules engine -- a set of rules for applying the knowledge base to situations provided to the program.

EXAMPLES OF EXPERT SYSTEM:
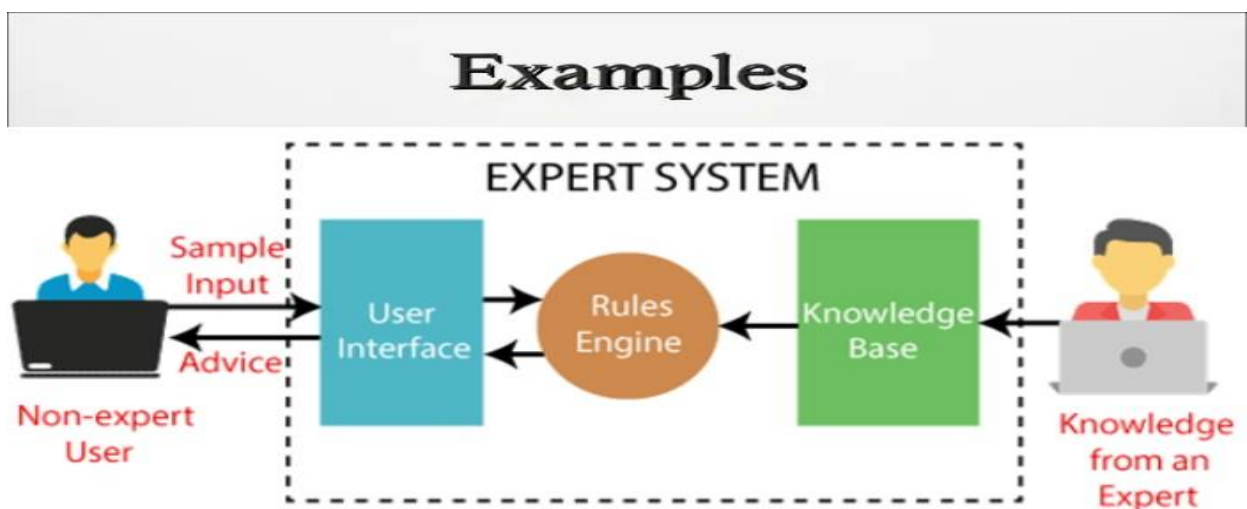
APPLICATIONS OF EXPERT SYSTEM:

Expert System in Education : In the field of education, many of the expert system's application are embedded inside the Intelligent Tutoring System (ITS) by using techniques from adaptive hypertext and hypermedia.

Expert system in Agriculture : The expert system for agriculture is same as like other fields. Here also the expert system uses the rule based structure and the knowledge of a human expert is captured in the form of IF-THEN rules and facts which are used to solve problems by answering questions typed at a keyboard attached to a computer.

Other examples may be MYCIN, DENDRAL, and CALEX etc.

Expert System for a particular decision problem : The expert system can be used as a stand alone advisory system for the specific knowledge domain. It also can provide decision support for a high level human expert.

Expert System for Text Animation (ESTA) : The idea behind creating an expert system is that it can enable many people to benefit from the knowledge of one person – the expert. By providing it with a knowledge base for a certain subject area, ESTA can be used to create an expert system for the subject:

ESTA + Knowledge base = Expert System

IBM Watson :IBM is a **data analytics processor** that uses natural language processing, a technology that analyzes human speech for meaning and syntax. IBM Watson performs analytics on vast repositories of data that it processes to answer human-posed questions, often in a fraction of a second.

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A Text to Speech (TTS) program is a software application that converts written text into spoken words. This technology uses natural language processing and synthesis to interpret and articulate the text in a human-like voice. TTS programs can be used for various purposes, including accessibility for individuals with visual impairments, language learning, audiobook creation, and hands-free operation of devices. The program typically allows users to customize aspects such as voice style, pitch, and speed to suit their preferences. With advances in artificial intelligence and deep learning, modern TTS systems can produce remarkably realistic and expressive speech, enhancing the user experience across different applications and devices.

ALGORITHM:

Step 1: Start the Program.

Step 2 : Prompt the user to enter the text they want to convert to speech.

Step 3 : Use the gTTS library to convert the text input into speech.

Step 4 :Specify the language (e.g., 'en' for English) and optionally the desired speed and pitch.

Step 5 : Save the synthesized speech to an audio file (e.g., MP3) or play it directly using a media player.

Step 6 : Stop the Program.

PROGRAM:

pip install gTTS

```python
# Import the required module for text
# to speech conversion
from gtts import gTTS


# This module is imported so that we can
# play the converted audio
import os


# The text that you want to convert to audio
mytext = 'Welcome to Python!'


# Language in which you want to convert
language = 'en'


# Passing the text and language to the engine,
# here we have marked slow=False. Which tells
# the module that the converted audio should
# have a high speed
myobj = gTTS(text=mytext, lang=language, slow=False)


# Saving the converted audio in a mp3 file named
# welcome
myobj.save("welcome.mp3")


# Playing the converted file
os.system("start welcome.mp3")
```

OUTPUT:

```
The output of the above program should be a
voice saying, 'Welcome to Python!'
```

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	**MAX. MARKS**	**MARKS AWARDED**
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A Translator program is a tool designed to convert text or speech from one language into another. Using sophisticated algorithms and linguistic models, these programs analyze input text or speech, identify the language, and then generate accurate translations in the desired target language. Translator programs play a vital role in breaking down language barriers, facilitating communication, and promoting cross-cultural understanding. They are used for various purposes, including travel, international business, academic research, and personal correspondence. Modern translation programs often incorporate artificial intelligence and machine learning techniques to improve translation quality and efficiency over time. Users can access these programs through websites, mobile apps, and integrated software solutions, making language translation more accessible and convenient in our interconnected world.

ALGORITHM:

Step 1: Start the program.

Step 2: Prompt the user to enter the text they want to translate.

Step 3: Ask for the target language to which the text should be translated.

Step 4: Use the googletrans library to translate the input text into the specified target language.

Step 5: Print the translated text to the user.

Step 6:Stop the program.

PROGRAM:

```
From googletrans import Translator, LANGUAGES
        def translate_text(text, dest_language):
```

```python
                    translator = Translator()
    translation = translator.translate(text, dest=dest_language)
    return translation.text


# Example usage:
def main():
    text = input("Enter the text to translate: ")


    # Print available languages for user reference
    print("Available Languages:")
    for code, language in LANGUAGES.items():
        print(f"{code}: {language}")


    dest_language = input("Enter the destination language code (e.g., 'ta' for Tamil, 'fr' for French): ")


    if dest_language not in LANGUAGES:
        print("Invalid language code. Please try again.")
        return


    translated_text = translate_text(text, dest_language)


    print("Original Text:", text)
    print("Translated Text:", translated_text)


if _name_ == "_main_":
    main()
```

OUTPUT:

```
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.


Original Text: Hello,How are you
Translated Text: Bonjour comment allez-vous
```

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A word-to-synonyms program is a linguistic tool designed to provide users with alternative words or phrases that have similar meanings to a given input word. Utilizing various algorithms and language databases, these programs analyze semantic relationships between words to generate a list of synonyms, enabling writers, students, and language enthusiasts to enrich their vocabulary and diversify their language usage. These programs often employ techniques such as natural language processing (NLP) to understand context and select appropriate synonyms that fit the intended meaning. Additionally, some advanced versions may offer features like antonyms, contextual usage examples, and related words to further enhance the user's understanding and expression. Whether used for writing, educational purposes, or simply expanding one's linguistic horizons, word-to-synonyms programs serve as valuable aids in communication and language enrichment.

ALGORITHM:

Step 1: Prompt the user to enter a word for which they want to find synonyms.

Step 2 : Construct a request URL to the thesaurus API (e.g., Datamuse API, WordsAPI) with the user's input word.

Step 3: Use the requests library to send a GET request to the thesaurus API.

Step 4 : Parse the JSON response received from the API.

Step 5 : Extract synonyms from the response data.

Step 6 : Print the list of synonyms to the user.

PROGRAM:

import nltk

from nltk.corpus import wordnet

```python
# Download WordNet data
nltk.download('wordnet')


def find_synonyms(word):
    synonyms = []
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.append(lemma.name())
    return synonyms


def main():
    word = input("Enter a word: ")
    synonyms = find_synonyms(word)
    if synonyms:
        print(f"Synonyms of '{word}':")
        print(synonyms)
    else:
        print("No synonyms found.")


if _name_ == "_main_":
    main()
```

OUTPUT:

```
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
Enter the Word:Speak
["Utter","Voice","Express","Articulate","Verbalite"]
```

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A grammar check program is a linguistic tool designed to identify and correct grammatical errors in written text. It employs algorithms that scan text for syntax, punctuation, and spelling mistakes to enhance overall clarity and coherence. These programs often utilize natural language processing (NLP) techniques to understand context and provide accurate corrections. They can detect various grammatical issues, including subject-verb agreement errors, misplaced modifiers, and improper verb tense usage. Additionally, grammar checkers may offer suggestions for improving sentence structure and readability. Some advanced versions integrate style guidelines to help users adhere to specific writing conventions or standards. From casual correspondence to professional documents, grammar check programs play a crucial role in refining written communication and promoting linguistic accuracy. These programs can also assist users in learning and reinforcing grammatical rules by providing explanations for the detected errors. With the ability to analyze text quickly and comprehensively, grammar check programs save time and effort in the proofreading process.

ALGORITHM:

Step 1: Prompt the user to enter the text they want to check for grammar errors.

Step 2: Use the language_tool_python library to analyze the input text for grammar and stylistic errors.

Step 3: Send the text to LanguageTool API for analysis.

Step 4: Parse the response from LanguageTool to identify grammar errors and suggestions.

Step 5 :Display the identified errors and suggested corrections to the user.

PROGRAM:

importing the package

import language_tool_python

```python
# using the tool
my_tool = language_tool_python.LanguageTool('en-US')


# given text
my_text = """LanguageTool provides utility to check grammar and spelling errors. We just have to paste
the text here and click the 'Check Text' button. Click the colored phrases for for information on potential
errors. or we can use this text too see an some of the issues that LanguageTool can dedect. Whot do
someone thinks of grammar checkers? Please not that they are not perfect. Style problems get a blue
marker: It is 7 P.M. in the evening. The weather was nice on Monday, 22 November 2021"""


# getting the matches
my_matches = my_tool.check(my_text)


# defining some variables
myMistakes = []

myCorrections = []

startPositions = []

endPositions = []


# using the for-loop
for rules in my_matches:

    if len(rules.replacements) > 0:

        startPositions.append(rules.offset)

        endPositions.append(rules.errorLength + rules.offset)

        myMistakes.append(my_text[rules.offset : rules.errorLength + rules.offset])

        myCorrections.append(rules.replacements[0])


# creating new object
my_NewText = list(my_text)


# rewriting the correct passage
for n in range(len(startPositions)):

    for i in range(len(my_text)):
```

```python
            my_NewText[startPositions[n]] = myCorrections[n]
        if (i > startPositions[n] and i < endPositions[n]):
            my_NewText[i] = ""


my_NewText = "".join(my_NewText)


# printing the text
print(my_NewText)
```

OUTPUT:

```
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36)
[MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
LanguageTool provides utility to check grammar and spelling
errors. We just have to paste the text here and click the '
Check Text' button. Click the colored phrases for informati
on on potential errors. Or we can use this text to see a so
me of the issues that LanguageTool can detect. Who do someo
ne thinks of grammar checkers? Please note that they are no
t perfect. Style problems get a blue marker: It is 7 P.M..
The weather was nice on Monday, 22 November 2021
```

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	**MAX. MARKS**	**MARKS AWARDED**
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT:

Ex.No:	
Date:	

AIM:

DESCRIPTION:

A text-to-image program is a powerful tool that converts textual input into visual representations or images. By leveraging advanced algorithms and deep learning techniques, these programs can interpret and translate written descriptions, prompts, or commands into graphical content. They often utilize neural networks to understand the semantics and context of the provided text, enabling the generation of relevant and meaningful images. Text-to-image programs find applications in various fields, including graphic design, creative arts, and data visualization. They can be used to generate illustrations, diagrams, or visualizations based on textual descriptions or concepts, offering a versatile solution for visual storytelling and communication. These programs enable users to create imagery from textual content rapidly, bypassing the need for manual drawing or design skills. With continuous advancements in artificial intelligence and image processing, text-to-image programs are becoming increasingly sophisticated, capable of producing high-quality and realistic visual outputs that cater to diverse creative and practical needs.

ALGORITHM:

Step 1: Start with the text you want to turn into an image.

Step 2: Decide on the dimensions (width and height) for your output image.

Step 3: Use a graphics library (like PIL/Pillow in Python) to create a blank image with the chosen dimensions.

Step 4: Choose a font, size, color, and alignment for your text.

Step 5: Use the graphics library to add your text to the image canvas, specifying the text content, position, font, and color.

Step 6: Save the resulting image to a file or display it directly.

PROGRAM:

1.Import libraries:

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.utils import save_image
import numpy as np
```

2.Define the generator and Discriminator:

```python
# Generator
class Generator(nn.Module):
    def __init__(self, latent_dim, img_shape):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512, 0.8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, np.prod(img_shape)),
            nn.Tanh()
        )
        self.img_shape = img_shape
    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), *self.img_shape)
        return img
# Discriminator
```

```python
class Discriminator(nn.Module):

def __init__(self, img_shape):

super(Discriminator, self).__init__()

self.model = nn.Sequential(

nn.Linear(np.prod(img_shape), 512),

nn.LeakyReLU(0.2, inplace=True),

nn.Linear(512, 256),

nn.LeakyReLU(0.2, inplace=True),

nn.Linear(256, 1),

nn.Sigmoid()

)

self.img_shape = img_shape

def forward(self, img):

img_flat = img.view(img.size(0), -1)

validity = self.model(img_flat)

return validity
```

3.Initialize tha GAN,Optimizers,and Loss functions:

```python
# Initialize parameters

latent_dim = 100

img_shape = (1, 28, 28) # FashionMNIST image

shape

# Initialize generator and discriminator

generator = Generator(latent_dim, img_shape)

discriminator = Discriminator(img_shape)

# Define loss function and optimizers

adversarial_loss = nn.BCELoss()

optimizer_G =

optim.Adam(generator.parameters(), lr=0.0002,

betas=(0.5, 0.999))

optimizer_D =
```

```python
optim.Adam(discriminator.parameters(),

lr=0.0002, betas=(0.5, 0.999))
```

4.Training the GAN:

```python
# Training

epochs = 200

batch_size = 64

sample_interval = 400

# Load FashionMNIST dataset

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,))

])

dataloader = torch.utils.data.DataLoader(

datasets.FashionMNIST(

"../data/fashionmnist",

download=True,

transform=transform,

),

batch_size=batch_size,

shuffle=True,

)

for epoch in range(epochs):

for i, (imgs, _) in enumerate(dataloader):

# Adversarial ground truths

valid = torch.ones(imgs.size(0), 1)

fake = torch.zeros(imgs.size(0), 1)

# -----------------

# Train Generator

# -----------------

optimizer_G.zero_grad()
```

```python
# Sample noise
z = torch.randn(imgs.size(0), latent_dim)
# Generate a batch of images
gen_imgs = generator(z)
# Generator loss
g_loss =
adversarial_loss(discriminator(gen_imgs), valid)
# Backpropagation
g_loss.backward()
optimizer_G.step()
# --------------------
# Train Discriminator
# --------------------
optimizer_D.zero_grad()
# Discriminator loss on real images
real_loss =
adversarial_loss(discriminator(imgs), valid)
# Discriminator loss on fake images
fake_loss =
adversarial_loss(discriminator(gen_imgs.detach()),
fake)
# Total discriminator loss
d_loss = (real_loss + fake_loss) / 2
# Backpropagation
d_loss.backward()
optimizer_D.step()
if i % sample_interval == 0:
print(
"[Epoch %d/%d] [Batch %d/%d] [D loss:
%f] [G loss: %f]"
```

```python
            % (epoch, epochs, i, len(dataloader),
            d_loss.item(), g_loss.item())
            )

            batches_done = epoch * len(dataloader) + i
            if batches_done % sample_interval == 0:
                save_image(gen_imgs.data[:25],
                "images/%d.png" % batches_done, nrow=5,
                normalize=True)
```

OUTPUT:

Given input: Tree

Conversion:

PAAVAI ENGINEERING COLLEGE (Autonomous)		
DESCRIPTION	MAX. MARKS	MARKS AWARDED
Preparation & Conduction	10	
Observation & Results	20	
Record Completion	05	
Viva Voce	05	
TOTAL	40	

RESULT: