List of Experiments:

1. Study of PROLOG. Write the following programs using PROLOG. (CO1)

2. Write a program to solve 8 queens problem. (CO3)

3. Solve any problem using depth first search. (CO2)

4. Solve any problem using best first search. (CO2)

5. Solve 8-puzzle problem using best first search. (CO2)

6. Solve Robot (traversal) problem using means End Analysis. (CO3)

7. Solve traveling salesman problem. CO3)

Department of Computer Science and Engineering

INDEX

SR NO	**NAME OF EXPERIMENT**	DATE OF PERFORMANCE	DATE OF SUBMISSION	REMARK	SIGNATURE
1	Study of PROLOG. Write the following programs using PROLOG				
2	Write a program to solve 8 queens problem.				
3	Solve any problem using depth first search				
4	Solve any problem using best first search				
5	Solve 8-puzzle problem using best first search				
6	Solve Robot (traversal) problem using means End Analysis				
7	Solve traveling salesman problem				

EXPERIMENT NO 1

AIM: Study of PROLOG. Write the following programs using PROLOG

Prolog stands for programming in logic. In the logic programming paradigm, prolog language is most widely available. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. A logical relationship describes the relationships which hold for the given application.

To obtain the solution, the user asks a question rather than running a program. When a user asks a question, then to determine the answer, the run time system searches through the database of facts and rules.

Prolog features are 'Logical variable', which means that they behave like uniform data structure, a backtracking strategy to search for proofs, a pattern-matching facility, mathematical variable, and input and out are interchangeable.

To deduce the answer, there will be more than one way. In such case, the run time system will be asked to find another solution. To generate another solution, use the backtracking strategy. Prolog is a weakly typed language with static scope rules and dynamic type checking.

Prolog is a declarative language that means we can specify what problem we want to solve rather than how to solve it.

Prolog is used in some areas like database, natural language processing, artificial intelligence, but it is pretty useless in some areas like a numerical algorithm or instance graphics.

In artificial intelligence applications, prolog is used. The artificial intelligence applications can be automated reasoning systems, natural language interfaces, and expert systems. The expert system consists of an interface engine and a database of facts..

A basic logic programming environment has no literal values. An identifier with upper case letters and other identifiers denote variables. Identifiers that start with lower-case letters denote data values. The basic Prolog elements are typeless. The most implementations of prolog have been enhanced to include integer value, characters, and operations. The Mechanism of prolog describes the tuples and lists.

Functional programming language and prolog have some similarities like Hugs. A logic program is used to consist of relation definition. A functional programming language is used to consist of

a sequence of function definitions. Both the logical programming and functional programming rely heavily on recursive definitions.

Applications of Prolog

The applications of prolog are as follows:

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

Procedure to run Prolog

Using the built-in predicates, the sequence of goals, or specifying a goal at the system prompt would be of little value in itself. To write a Prolog program, firstly, the user has to write a program which is written in the Prolog language, load that program, and then specify a sequence of one or more goals at the prompt.

To create a program in Prolog, the simple way is to type it into the text editor and then save it as a text file like prolog1.pl.

Following example shows a simple program of Prolog. The program contains three components, which are known as clauses. Each clause is terminated using a full stop.

1. dog(rottweiler).
2. cat(munchkin).
3. animal(A) :- cat(A).
4. Using the built-in predicate 'consult', the above program can be loaded in the Prolog system.
5. ?-consult('prolog1.pl').
6. This shows that prolog1.pl file exists, and the prolog program is systemically correct, which means it has valid clauses, the goal will succeed, and to confirm that the program has been correctly read, it produces one or more lines of output. e.g.,

EXPERIMENT NO 2

Aim : Write a program to solve 8 queens problem.

This exercise develops the skill of representing and manipulating structured data objects. It also illustrates Prolog as a natural database query language. A database can be naturally represented in Prolog as a set offact

The three solutions to the eight queens problem show how the same problem can be approached in different ways. we also varied the representation of data. Sometimes the representation was more economical, sometimes it was more explicit and partially redundant. The drawback of the more economical repre sentation is that some information always has to be recomputed when it is required

First solution

First we have to choose a representation of the board position. One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen. Each item in the list will specify a square of the board on which the corresponding queen is sitting. Further, each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8. In the program we can write such a pair as X/Y where, of course, the '/' operator is not meant to indicate division, combines both coordinates together into a square. Figure solution of the eight queens problem and its list representation. Having chosen this representation, the problem is to find such a list of the form Ix1/Y1, X2N2, X3/Y3, ..., X8/Y8l which satisfies the no-attack requirement. Our procedure solution will have to search for a proper instantiation of the variables Xl, Y l, Xz,Y 2, . .., X8, Y8. As we know that all the queens will have to be in different columns to prevent vertical attacks, we can immediately constrain the choice and so make the search task easier. We can thus fix the X-coordinates so that the solution list will fit the following, more specific template: form [1/Y1 ,2N2,3N3, ..., 8/Y8l I 8 This position can be specified by Figure the list Ul4, but simply shows one

8						*		
7			*					
6					*			
5							*	
4	*							
3				*				
2		*						
1								*
	1	2	3	4	5	6	7	8

A solution to the eight queens problem. This position can be specified by the list (1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1)

Second solution

In the board representation of solution 1 , each solution had the form [1/Y1, 2N2,3N3,..., 8/Y8] Because the queens were simply placed in consecutive columns .No information is lost if the X-coordinates were omitted. So a more economical Representation of the board position can be used, retaining only the y-coordinatesof the queens: lYl,Y2, Y3, ..., Y8]

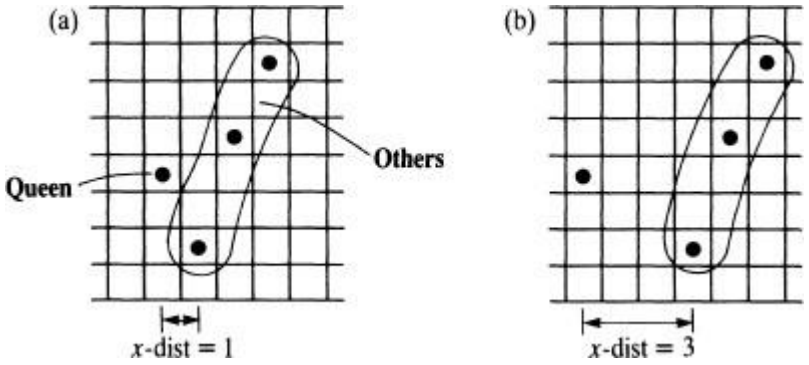
To prevent the horizontal attacks, no two queens can be in the same row. This imposes constraint n the Y-coordinates. The queens have to occupy all the rows L ,2, ...,8. The choice that remains is the order of these eight numbers.

Each solution is therefore represented by a permutation of the list

fL,2,3,4,5,6,7,91 Such a permutation, S, is a solution if all the queens are safe. So we can write:

solution(S) :-

permutation(f t,2,3,4,5,6,78, 1,S) safe(S).



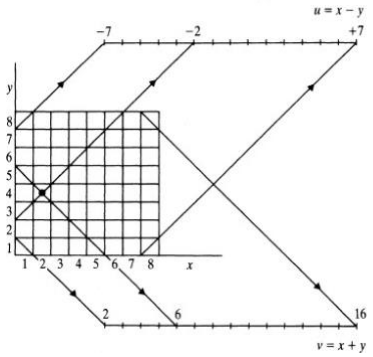
Solution 3

Third solution for the eight queen's problem will be based on the following Reasoning. Each queen has to be placed on some square; that is, into some Column, some row, some upward diagonal and some downward diagonal. To make sure that all the queens are safe, each queen must be placed in a different column, a different row, a different upward and a different downward diagonal. It is thus natural to consider a richer representations system with four coordinates:

X columns

Y rows

U upward diagonals



V downward diagonals

The relation between columns, rows, upward and downward diagonals.

The indicated square as coordinate 2,y = 4, u : 2 - 4 : -2, v : 2 + 4 : 6.

The coordinates are not independent: given x and y, u and v are determined

For example,a s

u=x-y
v=x+y
The domains for all four dimensions are:
Dx : [1,2,3,4,5,6,7,8]
Dy = [1,2,3,4,5,6,7,8]
Pu : [-7,- 6,- 5,-4,-3,-2,-1,0,1,2,3,4,5 ,6,7]
Dv : [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

Example of pseudo code for solving the 8 Queen Problem using backtracking:

```
N = 8 # (size of the chessboard)

def solveNQueens(board, col):
    if col == N:
        print(board)
        return True
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQueens(board, col + 1):
                return True
            board[i][col] = 0
    return False

def isSafe(board, row, col):
    for x in range(col):
        if board[row][x] == 1:
            return False
    for x, y in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    for x, y in zip(range(row, N, 1), range(col, -1, -1)):
        if board[x][y] == 1:
            return False
    return True

board = [[0 for x in range(N)] for y in range(N)]
if not solveNQueens(board, 0):
    print("No solution found")
```

sample output

```
[[1, 0, 0, 0, 0,  0, 0,  0], [0, 0, 0, 0, 0,  0, 1, 0], [0, 0, 0,  0,  1, 0,
0, 0], [0, 0,  0,  0, 0,  0, 0, 1], [0, 1, 0,  0, 0, 0, 0, 0], [0, 0,  0, 1,
0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0]
```

SAMPLE PYTHON PROGRAM FOR 8 QUEENS PROBLEM

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

EXPECTED OUTPUT

Enter the number of queens

8

Output:

```
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
```

EXPERIMENT NO 3

Aim: Solve any problem using depth first search

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal State.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.

Depth first search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses last in- first-out strategy and hence it is implemented using a stack.

Properties of depth-first (tree) search

• space complexity is O(bm) where b is the branching factor and m is the maximum depth of the tree

• time complexity is O(bm)

not complete (unless the state space is finite and contains no loops)— we may get stuck going down an infinite branch that doesn't lead to a solution

• even if the state space is finite and contains no loops, the first solution found by depth-first search may not be the shortest

Procedure to run depth first search in prolog

In order to test this code head over to Swish SWI prolog and paste this into terminal.

Then query the code and type on right hand side: solve(a, Sol)

```
% solve( Node, Solution):
%    Solution is an acyclic path (in reverse order) between Node and a goal
solve( Node, Solution)  :-
  depthfirst( [], Node, Solution).

% depthfirst( Path, Node, Solution):
%  extending the path [Node | Path] to a goal gives Solution

depthfirst( Path, Node, [Node | Path] ) :-
```

```
  goal( Node).

depthfirst( Path,Node,Sol) :-
 s( Node, Node1),
 \+ member(Node1, Path),          % Prevent a cycle
 depthfirst( [Node | Path], Node1, Sol).

depthfirst2( Node, [Node], _) :-
  goal( Node).

depthfirst2(Node,[Node|Sol],Maxdepth) :-
  Maxdepth > 0,
  s( Node, Node1),
  Max1 is Maxdepth - 1,
  depthfirst2( Node1, Sol, Max1).


goal(f).
goal(j).
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
s(c,g).
s(d,h).
s(e,i).
s(e,j).
```

The solution will be: Sol = [j, e, b, a]

You can debug this code by typing: trace, (solve(a, Sol)).

The solution will be: Sol = [f, c, a]

SAMPLE PYTHON PROGRAM FOR DEPTH FIRST SEARCH

```
# 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)

        # 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()
```

EXPERIMENT 4

AIM: Solve any problem using best first search

The best first search uses the concept of a priority queue and heuristic search. It is a search algorithm that works on a specific rule. The aim is to reach the goal from the initial state via the shortest path. 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.

Best First Search Algorithm

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the 'ordered' OPEN list
3. Repeat the next steps until the GOAL node is reached
    1. If the OPEN list is empty, then EXIT the loop returning 'False'
    2. 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
    3. 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
    4. 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
    5. 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).

Procedure to run best first search in prolog

```
// Pseudocode for Best First Search

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
```

SAMPLE PYTHON PROGRAM FOR BEST FIRST SEARCH

```
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:
                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)

Expected Output
0 1 3 2 8 9

EXPERIMENT NO 5

AIM: Solve 8-puzzle problem using best first search.

It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state".

Procedure:

INITIAL STATE

1	2	3
8		4
7	6	5

2	8	1
	4	3
7	6	5

GOAL STATE

The 8-puzzle problem belongs to the category of "sliding block puzzle" type of problem. The 8-puzzle i s a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it.

A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around

The control mechanisms for an 8-puzzle solver must keep track of the order in which operations are performed, so that the operations can be undone one at a time if necessary. The objective of the puzzles is to find a sequence of tile movements that leads from a starting configuration to a goal configuration such as two situations given below.

Figure (Starting State) (Goal State)

The state of 8-puzzle is the different permutation of tiles within the frame. The operations are the permissible moves up, down, left, right. Here at each step of the problem a function f(x) will be defined which is the combination of g(x) and h(x).

i.e. **F(x)=g(x) + h (x)**

Where

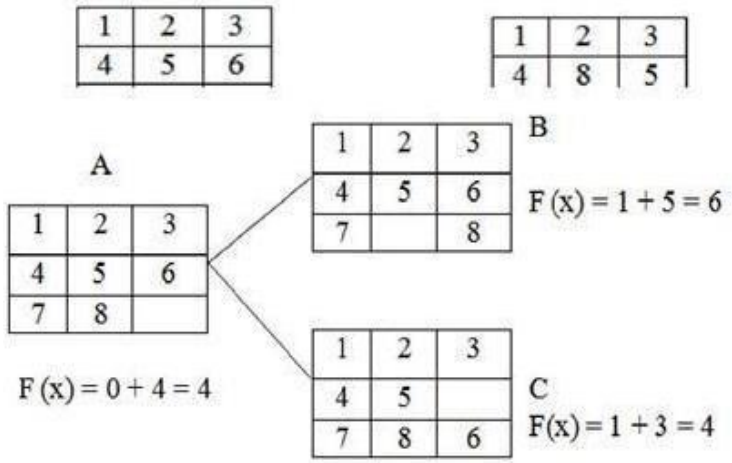
g (x): how many steps in the problem you have already done or the current state from the initial state.

h (x): Number of ways through which you can reach at the goal state from the current state or Or

h (x): is the heuristic estimator that compares the current state with the goal state note down how many states are displaced from the initial or the current state. After calculating the **f** value at each step finally take the smallest f (x) value at every step and choose that as the next current state to get the goal

Let us take an example.

Figure (Initial State) (Goal State)



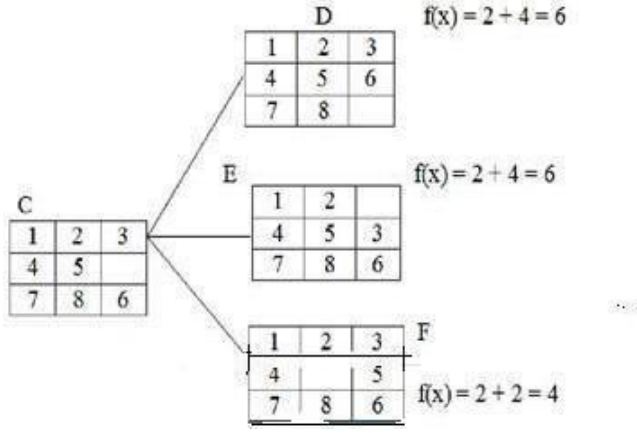
Step 1:

f (x) is the step required to reach at the goal state from the initial state. So in the trayeither 6 or 8 can change their portions to fill the empty position. So there will be two possible current states namely B and C. The f (x) value of B is 6 and that of C is 4. As 4 is the minimum, so take C as the current state to the next state.

Step 2:

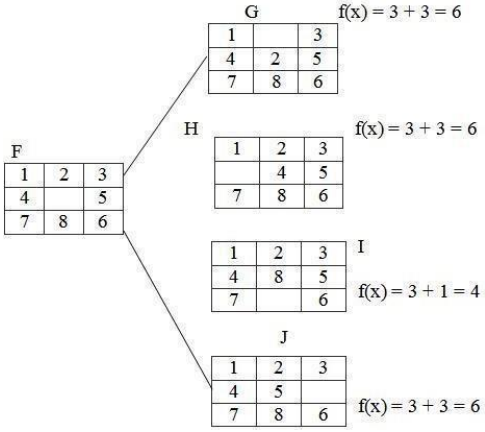
In this step, from the tray C three states can be drawn. The empty position will contain either 5 or 3 or 6. So for three different values three different states can be obtained. Then calculate each of their f (x) and take the minimum one.

Here the state F has the minimum value i.e. 4 and hence take that as the next current state.



Step 3:

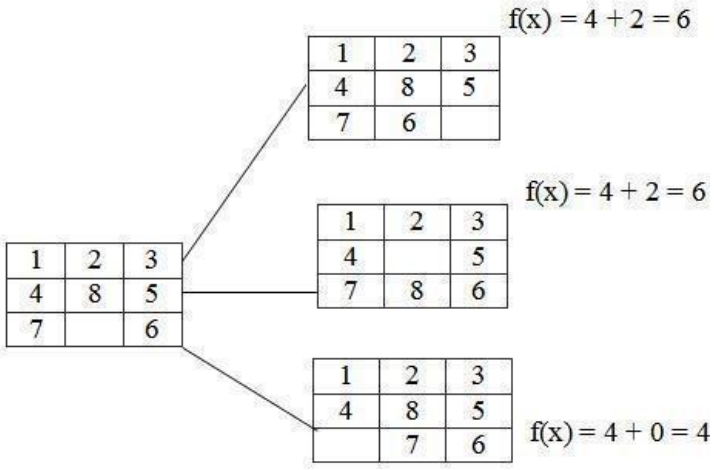
The tray F can have 4 different states as the empty positions can be filled with b4 values i.e.2, 4, 5, 8.



Step 4:

In the step-3 the tray I has the smallest f (n) value. The tray I can be implemented in 3 different states because the empty position can be filled by the members like 7, 8, 6.

Hence, we reached at the goal state after few changes of tiles in different positions of the trays.



SAMPLE PYTHON PROGRAM FOR 8 PUZZLE PROBLEM

```
# Python code to display the way from the root
# node to the final destination node for N*N-1 puzzle
# algorithm by the help of Branch and Bound technique
# The answer assumes that the instance of the
# puzzle can be solved
  # Importing the 'copy' for deepcopy method
import copy

# Importing the heap methods from the python
# library for the Priority Queue
from heapq import heappush, heappop

# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
# puzzle(n=4) and so on ...
n = 3

# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue
class priorityQueue:

    # Constructor for initializing a
    # Priority Queue
```

```
        def __init__(self):
            self.heap = []

    # Inserting a new key 'key'
    def push(self, key):
        heappush(self.heap, key)

    # funct to remove the element that is minimum,
    # from the Priority Queue
    def pop(self):
        return heappop(self.heap)

    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:

    def __init__(self, parent, mats, empty_tile_posi,
                 costs, levels):

        # This will store the parent node to the
        # current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent

        # Useful for Storing the matrix
        self.mats = mats

        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi

        # Store no. of misplaced tiles
        self.costs = costs

        # Store no. of moves so far
        self.levels = levels

    # This func is used in order to form the
    # priority queue based on
    # the costs var of objects
    def __lt__(self, nxt):
        return self.costs < nxt.costs

# method to calc. the no. of
# misplaced tiles, that is the no. of non-blank
```

```
# tiles not in their final posi
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1

    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
             levels, parent, final) -> nodes:

    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                      costs, levels)
    return new_nodes

# func to print the N by N matrix
def printMatsrix(mats):

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

        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):

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

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
```

```
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()

# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):

    # Creating a priority queue for storing the live
    # nodes of the search tree
    pq = priorityQueue()

    # Creating the root node
    costs = calculateCosts(initial, final)
    root = nodes(None, initial,
            empty_tile_posi, costs, 0)

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

    # Discovering a live node with min. costs,
    # and adding its children to the list of live
    # nodes and finally deleting it from
    # the list.
    while not pq.empty():

        # Finding a live node with min. estimatsed
        # costs and deleting it form the list of the
        # live nodes
        minimum = pq.pop()

        # If the min. is ans node
        if minimum.costs == 0:

            # Printing the path from the root to
            # destination;
            printPath(minimum)
            return

        # Generating all feasible children
        for i in range(n):
            new_tile_posi = [
                minimum.empty_tile_posi[0] + rows[i],
                minimum.empty_tile_posi[1] + cols[i], ]

            if isSafe(new_tile_posi[0], new_tile_posi[1]):

                # Creating a child node
```

```
            child = newNodes(minimum.mats,
                             minimum.empty_tile_posi,
                             new_tile_posi,
                             minimum.levels + 1,
                             minimum, final,)

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

# Main Code

# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
          [ 5, 6, 0 ],
          [ 7, 8, 4 ] ]

# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]

# Method call for solving the puzzle
solve(initial, empty_tile_posi, final)
```

Output:

```
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
```

EXPERIMENT NO 6

AIM: Solve Robot (traversal) problem using means End Analysis

- Means-Ends Analysis is problem-solving techniques used in Artificial intelligence for limiting search in AI programs.
- It is a mixture of Backward and forward search technique.
- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works:

The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving. Following are the main Steps which describes the working of MEA technique for solving a problem.

1. First, evaluate the difference between Initial State and final State.
2. Select the various operators which can be applied for each difference
3. Apply the operator at each difference, which reduces the difference between the current state and goal state.

Algorithm for Means-Ends Analysis:

Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.

- **Step 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.
    a. Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
    b. Attempt to apply operator O to CURRENT. Make a description of two states.
    i) O-Start, a state in which O?s preconditions are satisfied.
    ii) O-Result, the state that would result if O were applied In O-start.
    c. If
    (First-Part <------ MEA (CURRENT, O-START)

And

(LAST-Part < ---- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART

.

Solving the robot traversal problem using Means-End Analysis (MEA) in Python involves breaking down the problem into subgoals and finding a sequence of actions to reach the desired state. Here's a simplified example of how you might approach this problem:

SAMPLE PYTHON PROGRAM ON Robot (traversal) problem using means End Analysis

Let's say you have a grid world, and you want to move a robot from the starting position `(x1, y1)` to the target position `(x2, y2)`. The robot can move up, down, left, or right.

1. Define the initial state and the goal state.

```
python
initial_state = (x1, y1)
goal_state = (x2, y2)
```

2. Define a function to calculate the heuristic (how far the current state is from the goal) to guide the robot.

```
python
def heuristic(state):
    return abs(state[0] - goal_state[0]) + abs(state[1] - goal_state[1])
```

3. Implement a function to generate possible actions from the current state.

```
python
def generate_actions(state):
    x, y = state
    possible_actions = []

    # Check if the robot can move up
    if y > 0:
        possible_actions.append(("UP", (x, y - 1)))

    # Check if the robot can move down
    if y < max_y:
        possible_actions.append(("DOWN", (x, y + 1)))

    # Check if the robot can move left
    if x > 0:
```

```
        possible_actions.append(("LEFT", (x - 1, y)))

    # Check if the robot can move right
    if x < max_x:
        possible_actions.append(("RIGHT", (x + 1, y)))

    return possible_actions
```

4. Implement the Means-End Analysis algorithm to find the sequence of actions.

```
python
def means_end_analysis(start_state):
    path = []
    current_state = start_state

    while current_state != goal_state:
        possible_actions = generate_actions(current_state)
        # Sort actions based on heuristic (closest to goal first)
        possible_actions.sort(key=lambda x: heuristic(x[1]))

        # Choose the action that minimizes the heuristic
        action, new_state = possible_actions[0]

        # Add the chosen action to the path
        path.append(action)

        # Move to the new state
        current_state = new_state

    return path

# Call the function to find the sequence of actions
path_to_goal = means_end_analysis(initial_state)
print(path_to_goal)
```

EXPERIMENT NO 7

AIM: Solve traveling salesman problem

The Traveling Salesman Problem (TSP) is a classic optimization problem in AI and computer science. Its goal is to find the shortest possible route that visits a given set of cities and returns to the starting city. There are several algorithms to solve the TSP, and I'll describe two common approaches: Brute Force and Genetic Algorithms.

1. *Brute Force*:
   - Generate all possible permutations of cities.

- Calculate the total distance for each permutation.
- Select the permutation with the shortest distance.

While this method guarantees an optimal solution, it becomes impractical for a large number of cities due to the factorial complexity (n!). For even moderately sized instances, it's computationally infeasible.

Here's a simplified Python code snippet for the brute force approach:

```
python
from itertools import permutations

def calculate_total_distance(path, cities):
    total_distance = 0
    for i in range(len(path) - 1):
        total_distance += distance(cities[path[i]], cities[path[i + 1]])
    return total_distance

def brute_force_tsp(cities):
    num_cities = len(cities)
    best_path = None
    best_distance = float('inf')

    for path in permutations(range(num_cities)):
        current_distance = calculate_total_distance(path, cities)
        if current_distance < best_distance:
            best_distance = current_distance
            best_path = path

    return best_path, best_distance
```

2. *Genetic Algorithms*:
- Initialize a population of potential routes (individuals).
- Evaluate each route's fitness (shortness of the path).
- Select individuals for reproduction based on their fitness.
- Create new routes through crossover (combining routes) and mutation (small changes).
- Repeat the evaluation, selection, crossover, and mutation steps for several generations.
- The best route found after a certain number of generations is the solution.

Genetic algorithms can handle larger instances of the TSP more efficiently than brute force, although they may not always guarantee the optimal solution.

Here's a simplified Python code snippet for the genetic algorithm approach:

```
python
import random

def genetic_algorithm_tsp(cities, population_size=50, generations=1000):
```

```
    # Initialize population randomly
    population = [random.sample(range(len(cities)), len(cities)) for _ in
range(population_size)]

    for generation in range(generations):
        # Evaluate fitness of each individual
        fitness_scores = [(path, calculate_total_distance(path, cities)) for path in population]
        fitness_scores.sort(key=lambda x: x[1])

        # Select the top-performing individuals for reproduction
        selected = [path for path, _ in fitness_scores[:population_size // 2]]

        # Create new individuals through crossover and mutation
        children = crossover(selected)
        children = mutate(children)

        # Replace the old population with the new one
        population = selected + children

    # Find the best solution in the final population
    best_path, best_distance = min(fitness_scores, key=lambda x: x[1])
    return best_path, best_distance
```