

INDEX

S.No	DATE	TOPIC	Page No	SIGNATURE
1		Study of PROLOG Using Turbo Prolog	1	
2		Eight - Queens Problem	15	
3		Depth First Search	17	
4		Best First Search	19	
5		Robot Traversal Program using means End Analysis	22	
6		Travelling Salesman Problem	24	

Study of PROLOG Using Turbo Prolog

Topics:

- a) Basics of Turbo Prolog
- b) Intro to Prolog programming
- c) Running a simple program
 - Prolog is a logical programming language and stands for PROgramming in LOGic • Created around 1972
 - Preferred for AI programming and mainly used in such areas as:
 - o Theorem proving, expert systems, NLP, ...
 - Logical programming is the use of mathematical logic for computer programming. **MENU**
 - Files – Enables the user to load programs from disk, create new programs, save modified programs to disk, and to quit the program.
 - Edit – Moves user control to the Editor panel.
 - Run – Moves user control to the Dialog panel ; compiles the user program (if not already done so) in memory before running the program.
 - Compile – Provides the user with choices on how to save the compiled version of the program.
 - Options – Provides the user with choices on the type of compilation to be used. • Setup – Enables the user to change panel sizes, colors, and positions.

Dialog

When a Prolog program is executing, output will be shown in the Dialog Panel **Message**

The Message Panel keeps the programmer up to date on processing activity. **Trace**

The Trace Panel is useful for finding problems in the programs you create.

Prolog Clauses

- Any factual expression in Prolog is called a clause.
- There are two types of factual expressions: facts and rules
- There are three categories of statements in Prolog:
 - ♣ **Facts:** Those are true statements that form the basis for the knowledge base.
 - ♣ **Rules:** Similar to functions in procedural programming (C++, Java...) and has the form of if/then.
 - ♣ **Queries:** Questions that are passed to the interpreter to access the knowledge base and start the program.

What is a Prolog program?

Prolog is used for solving problems that involve objects and the relationships between objects.

- A program consists of a database containing one or more facts and zero or more rules(next week).
- A fact is a relationship among a collection of objects. A fact is a one-line statement that ends with a full-stop.
 - parent (john, bart).
 - parent (barbara, bart).
 - male (john).
 - dog(fido). >> Fido is a dog or It is true that fido is a dog
 - sister(mary, joe). >> Mary is Joe's sister.

play(mary, joe, tennis). >> It is true that Mary and Joe play tennis.

- Relationships can have any number of objects.
- Choose names that are meaningful – because in Prolog names are arbitrary strings but people will have to associate meaning to them

1. Program to demonstrate a simple prolog program.

predicates

like(symbol,symbol)

hate(symbol,symbol)

clauses

like(sita,ram).

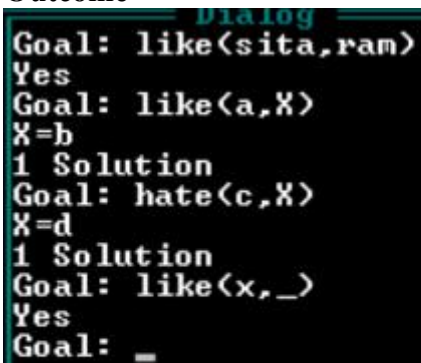
like(x,y).

like(a,b).

hate(c,d).

hate(m,n).

Outcome



```
Goal: like(sita,ram)
Yes
Goal: like(a,X)
X=b
1 Solution
Goal: hate(c,X)
X=d
1 Solution
Goal: like(x,_)
Yes
Goal: _
```

2. Program to add two numbers.

predicates

add

clauses

```
add:-write("input first number"), readint(X),  
write("input second number"), readint(Y),  
Z=X+Y,write("output=",Z).
```

Outcome :



```
Goal: add  
input first number4  
input second number7  
output=11Yes  
Goal: _
```

3. Program to categorise animal characteristics.

predicates

small(symbol)

large(symbol)

color(symbol,symbol)

clauses

small(rat).

small(cat).

large(lion).

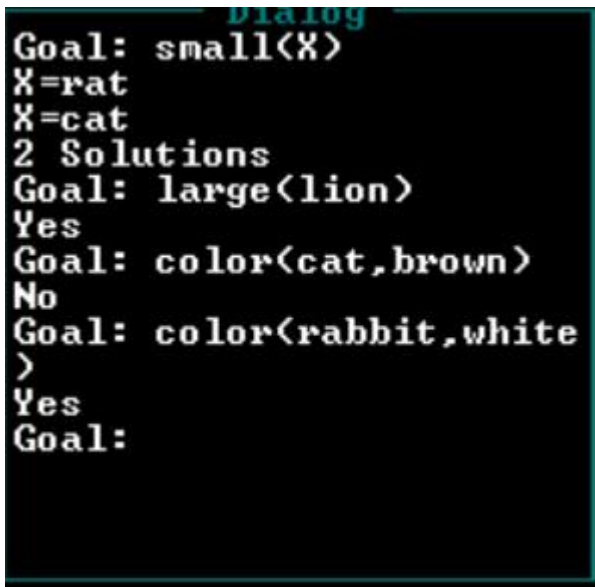
color(dog,black).

color(rabbit,white).

color(X,dark):-

color(X,black);color(X,brown)

Outcome:



```
Goal: small(X)
X=rat
X=cat
2 Solutions
Goal: large(lion)
Yes
Goal: color(cat,brown)
No
Goal: color(rabbit,white)
Yes
Goal:
```

4. (a) Program to count number of elements in a list .

domains

x=integer

list=integer*

predicates

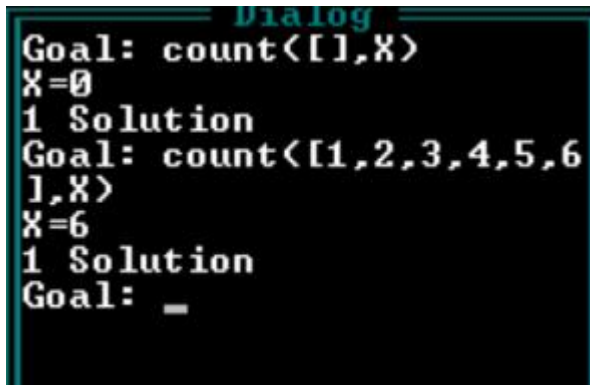
count(list,x)

clauses

count([],0).

count([_|T],N):-count(T,N1),N=N1+1.

Outcome :



```
Dialog
Goal: count([1,X)
X=0
1 Solution
Goal: count([1,2,3,4,5,6
1,X)
X=6
1 Solution
Goal: _
```

4. (b) Program to reverse the list .

domains

x=integer

list=integer*

predicates

append(x,list,list)

rev(list,list)

clauses

append(X,[],[X]).

append(X,[H|T],[H|T1]):-append(X,T,T1). rev([],[]).

rev([H|T,rev):-rev(T,L),append(H,L,rev).

Outcome :

```
Goal: append(2,[3,4,5],X)
>
X=[3,4,5,2]
1 Solution
Goal: rev([1,2,3,4],X)
X=[4,3,2,1]
1 Solution
Goal:
```


4. (c) Program to replace an integer from the list .

domains

list=integer*

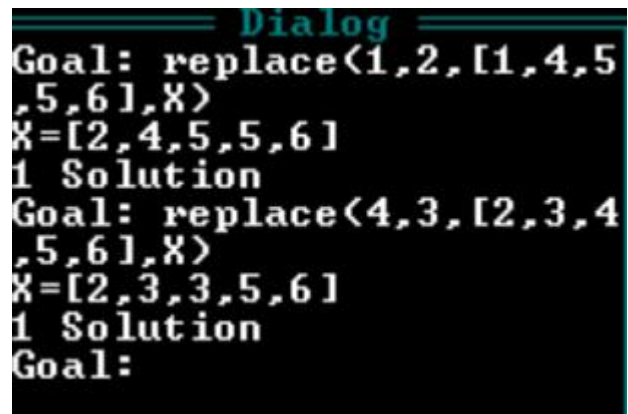
predicates

replace(integer,integer,list,list) **clauses**

replace(X,Y,[X|T],[Y|T]).

replace(X,Y,[H|T],[H|T1]):-replace(X,Y,T,T1).

Outcome:



```
Dialog
Goal: replace(1,2,[1,4,5,5,6],X)
X=[2,4,5,5,6]
1 Solution
Goal: replace(4,3,[2,3,4,5,6],X)
X=[2,3,3,5,6]
1 Solution
Goal:
```

4. (d) Program to delete an integer from the list .

domains

list=integer*

predicates

del(integer,list,list)

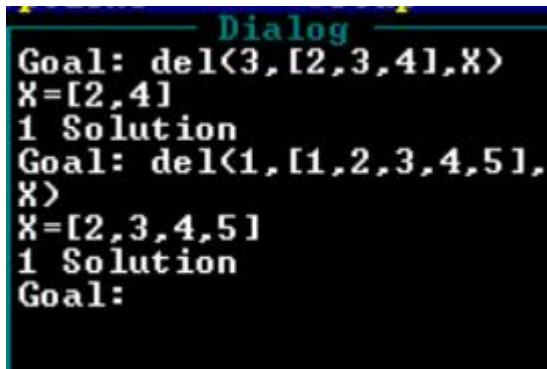
clauses

del(X,[X|T],T).

del(X,[H|T],[H|T1]):-

del(X,T,T1).

Outcome:



```
Dialog
Goal: del(3,[2,3,4],X)
X=[2,4]
1 Solution
Goal: del(1,[1,2,3,4,5],X)
X=[2,3,4,5]
1 Solution
Goal:
```

5. Write a program to classify diseases using prolog.

domains

disease,indication = symbol

predicates

symptom(disease, indication)

clauses

symptom(chicken_pox, high_fever).

symptom(chicken_pox, chills).

symptom(flu, chills).

symptom(cold, mild_body_ache).

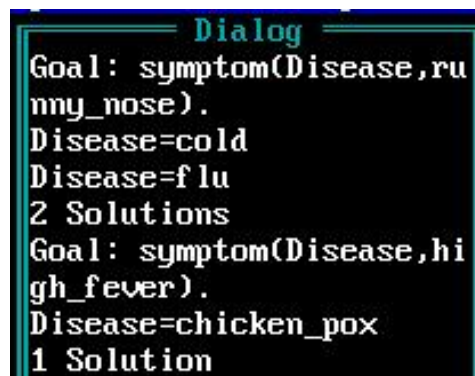
symptom(flu, severe_body_ache).

symptom(cold, runny_nose).

symptom(flu, runny_nose).

symptom(flu, moderate_cough).

Outcome:

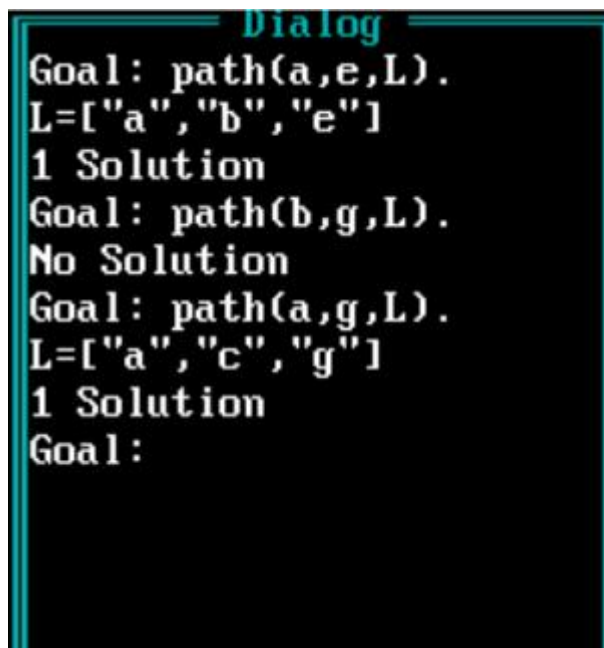


```
Dialog
Goal: symptom(Disease,runny_nose).
Disease=cold
Disease=flu
2 Solutions
Goal: symptom(Disease,high_fever).
Disease=chicken_pox
1 Solution
```

6. Write a program of depth first search

```
domains
X=symbol
Y=symbol*
predicates
child(X,X)
childnode(X,X,Y)
path(X,X,Y)
clauses
child(a,b). /*b is child of a*/
child(a,c). /*c is child of a*/
child(a,d). /*d is child of a*/
child(b,e). /*b is child of b*/
child(b,f). /*f is child of b*/
child(c,g). /*g is child of c*/
path(A,G,[A|Z]):- /*to find the path from root to leaf*/
childnode(A,G,Z).
childnode(A,G,[G]):- /*to determine whether a node is child of
other*/
child(A,G).
childnode(A,G,[X|L]):-
child(A,X),
childnode(X,G,L).
```

Outcomes:



```
Dialog
Goal: path(a,e,L).
L=["a","b","e"]
1 Solution
Goal: path(b,g,L).
No Solution
Goal: path(a,g,L).
L=["a","c","g"]
1 Solution
Goal:
```

7. Write a program to solve traveling salesman problem.

Domains

town = symbol

distance = integer

predicates

nondeterm road(town, town, distance)

nondeterm route(town, town, distance)

clauses

road("coimbatore", "madurai", 950).

road("trichy", "madurai", 750).

road("coimbatore", "trichy", 250).

road("trichy", "chennai", 300).

road("coimbatore", "chennai", 500).

route(Town1, Town2, Distance) :-

road(Town1, Town2, Distance).

route(Town1, Town2, Distance) :-

road(Town1, X, Dist1),

route(X, Town2, Dist2),

Distance = Dist1 + Dist2.

Outcome:

```
Goal: route("trichy","ma
durai",X),write("distanc
e from trichy to madurai
is",X),nl.
distance from trichy to
madurai is750
X=750
1 Solution
Goal: S_
```

8.Program to read address of a person using compound variable .

domains

person=address(name,street,city,state,zip)

name,street,city,state,zip=String

predicates

readaddress(person)

go

clauses

go:-

readaddress(Address),nl,write(Address),nl,nl,
write("Accept(y/n)?"),readchar(Reply),Reply='y',!.

go:-

nl,write("please re-enter"),nl,go.

readaddress(address(N,street,city,state,zip)):-

write("Name:"),readln(N),

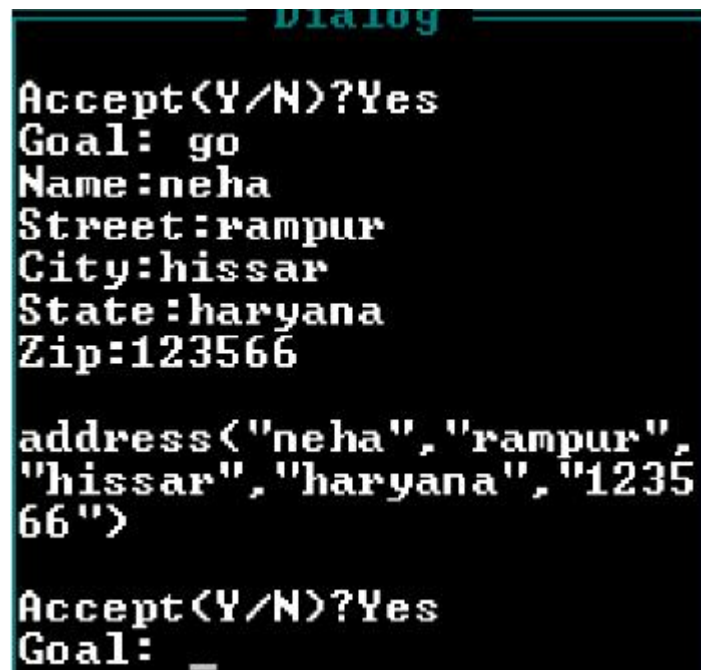
write("Street:"),readln(street),

write("City:"),readln(city),

write("State:"),readln(state),

write("Zip:"),readln(zip).

Outcome:



```
Accept<Y/N>?Yes
Goal: go
Name:neha
Street:rampur
City:hissar
State:haryana
Zip:123566

address("neha","rampur",
"hissar","haryana","1235
66")

Accept<Y/N>?Yes
Goal: _
```

9.Program to demonstrate family relationship

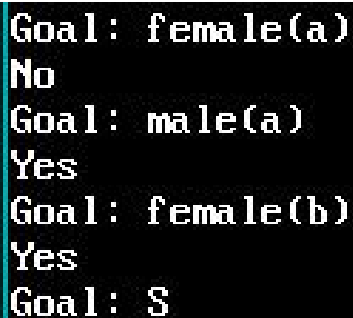
predicates

```
parent(symbol,symbol)
child(symbol,symbol)
mother(symbol,symbol)
brother(symbol,symbol)
sister(symbol,symbol)
grandparent(symbol,symbol)
male(symbol)
female(symbol)
```

clauses

```
parent(a,b).
sister(a,c).
male(a).
female(b).
child(X,Y):-parent(Y,X).
mother(X,Y):-female(X),parent(X,Y).
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
brother(X,Y):-male(X),parent(V,X),parent(V,Y).
```

Outcome:



```
Goal: female(a)
No
Goal: male(a)
Yes
Goal: female(b)
Yes
Goal: S
```

Result:

Thus the Prolog and programs of Prolog is studied and executed successfully .

Eight - Queens Problem

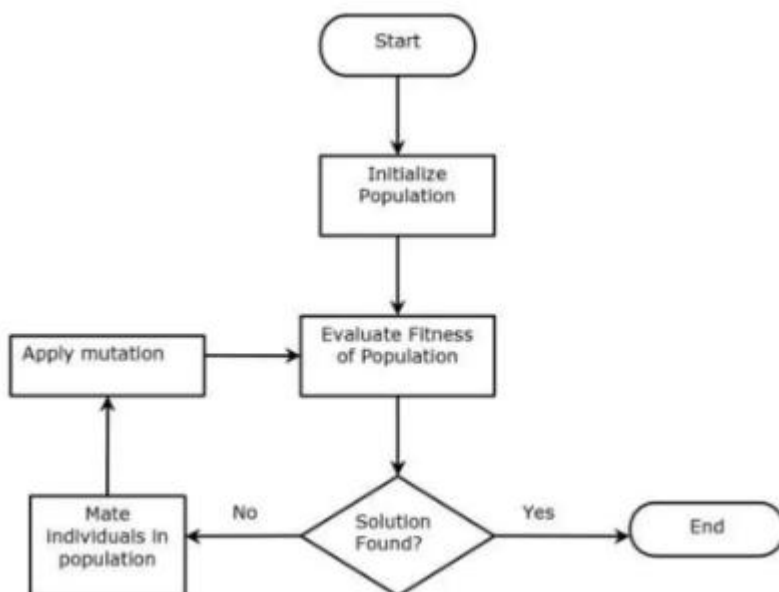
Aim :

To write a program to solve eight queens problem using python.

Algorithm:

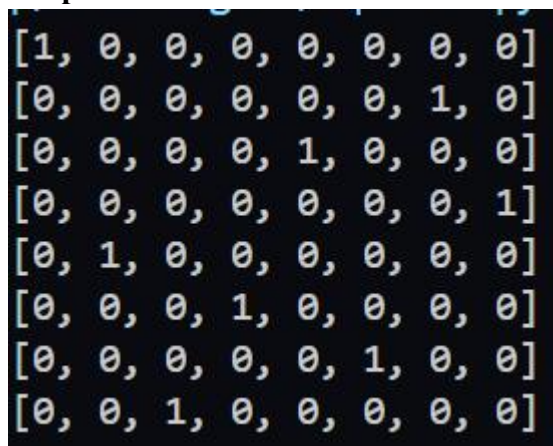
1. Initialize Board:
 - Create an 8x8 board with all entries set to 0.
2. Define isSafe(board, row, col):
 - Check Row: No queens in the same row.
 - Check Column: No queens in the same column.
 - Check Diagonals: No queens on the same diagonals.
3. Define solve Queens (board, col):
 - Base Case: If col == 8, print the board and return true.
 - For each row r in column col:
 - If isSafe(board, r, col):
 - Place queen at (r, col).
 - Recursively call solve Queens(board, col + 1).
 - If recursion returns true, return true.
 - Else: Backtrack (remove the queen) and try the next row.
4. Start Algorithm:
 - Call solves Queens (board, 0) to begin placing queens from the first column.

Flowchart:



Program:

```
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")
```

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]
```

Result:

Thus, the given 8 Queen python program was executed successfully and verified.

Depth First Search

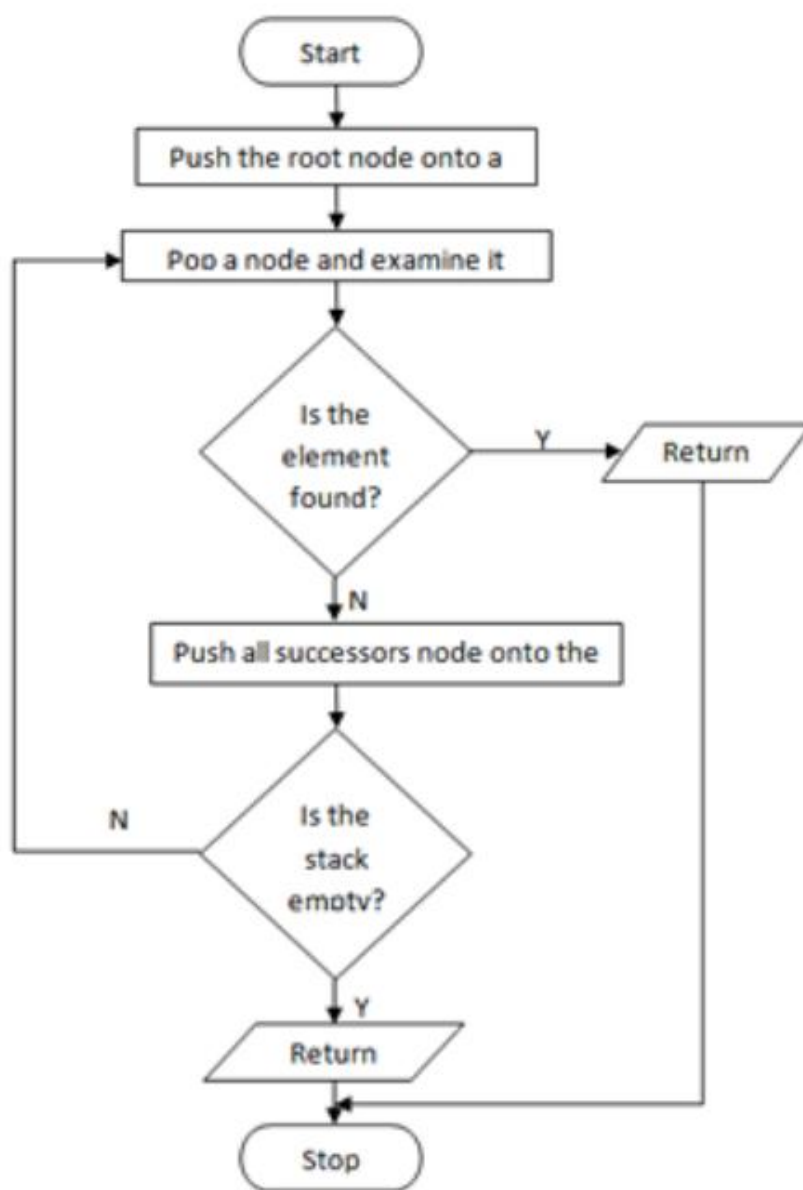
Aim:

To write a program to solve Depth First search using python.

Alogorthm:

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

Flowchart:



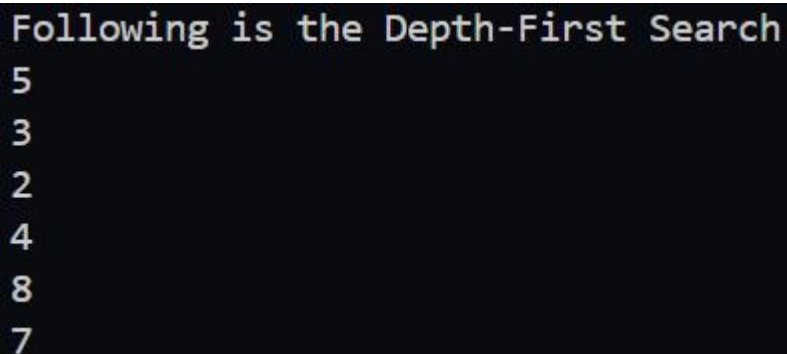
Program:

```
graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
    '7': ['8'],
    '2': [],
    '4': ['8'],
    '8': []
}

visited = set() # Set to keep track of visited nodes of the graph.

def dfs(visited, graph, node):
    # Function for DFS
    if node not in visited:
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

```
Following is the Depth-First Search
5
3
2
4
8
7
```

Result:

Thus a program to solve Depth First search using python is executed successfully and verified.

Best First Search

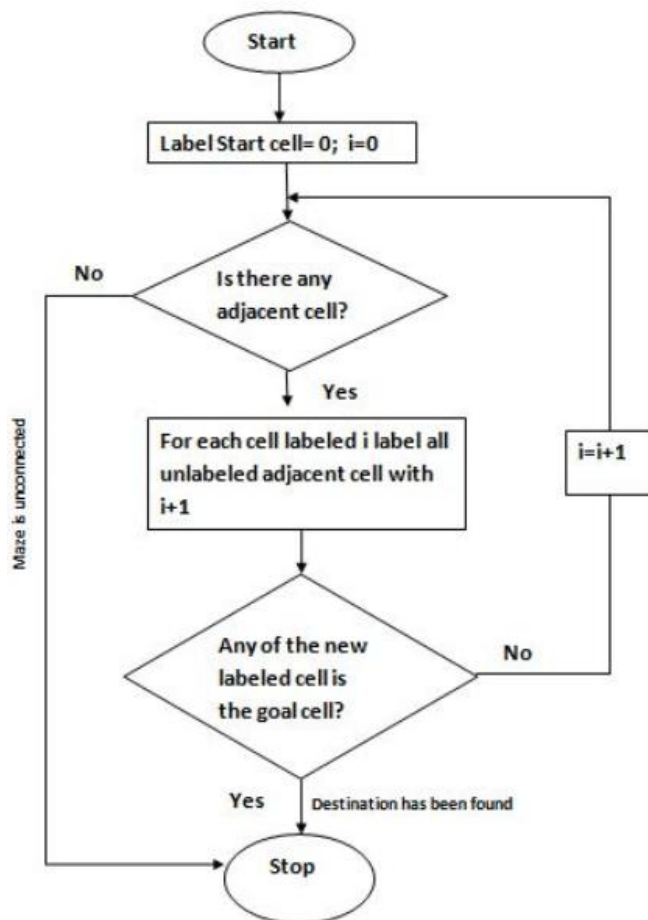
Aim:

To write a program to solve Best First search using python.

Algorithm:

- 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";

Flowchart :



Program:

```
from queue import PriorityQueue
# Number of vertices in the graph
v = 14
# Graph represented as an adjacency list
graph = [[] for i in range(v)]
# Function for implementing Best-First Search
# Outputs the path with the 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 not pq.empty():
        u = pq.get()[1]
        # Displaying the path with the lowest cost
        print(u, end=" ")
        if u == target:
            break
        # Explore all neighbors of u
        for v, c in graph[u]:
            if not visited[v]:
                visited[v] = True
                pq.put((c, v))
    print()
# Function to add edges to the graph
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
# Adding edges with their respective costs
addege(0, 1, 3)
addege(0, 2, 6)
addege(0, 3, 5)
addege(1, 4, 9)
addege(1, 5, 8)
addege(2, 6, 12)
addege(2, 7, 14)
addege(3, 8, 7)
addege(8, 9, 5)
addege(8, 10, 6)
addege(9, 11, 1)
addege(9, 12, 10)
addege(9, 13, 2)
# Define source and target nodes
source = 0
target = 9
```

```
# Execute the Best-First Search
print("Following is the Best-First Search path:")
best_first_search(source, target, v)
```

Output:

```
Following is the Best-First Search path:
0 1 3 2 8 9
```

Result:

Thus a program to solve Best First search using python is executed successfully and verified.

Robot Traversal Program using means End Analysis

Aim:

The aim of this program is to solve a robot traversal problem on a grid using Means-End Analysis (MEA).

Algorithm:

1. Initialize:

Define the starting position (x_start, y_start) and goal position (x_goal, y_goal) on a grid.
Initialize the robot's current position to the starting position.

2. Define Heuristic:

Calculate the heuristic distance between the current position and the goal

3. Identify Possible Moves:

Define possible moves as up, down, left, and right on the grid:

(x+1, y): Move right

(x-1, y): Move left

(x, y+1): Move up

(x, y-1): Move down

4. Move Selection:

For each possible move, calculate the heuristic distance to the goal.

Select the move with the minimum heuristic distance (i.e., the move that brings the robot closer to the goal).

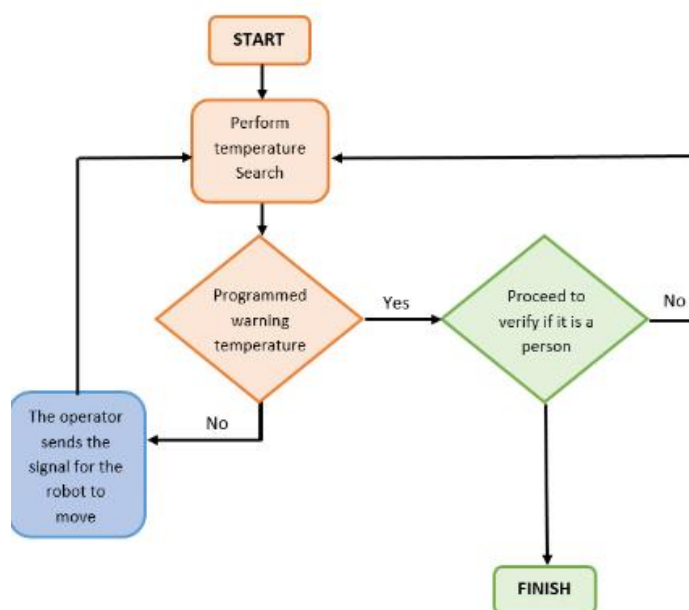
5. Update Position:

Update the robot's position to the selected move.

6. Repeat Until Goal is Reached:

Repeat steps 3-5 until the robot's current position matches the goal position.

Flowchart:



Program:

```
class RobotTraversal:
def __init__(self, start, goal):
self.start = start
self.goal = goal
self.current_position = start
def heuristic_distance(self, pos):
# Calculate Manhattan distance as the heuristic
return abs(self.goal[0] - pos[0]) + abs(self.goal[1] - pos[1])
def get_possible_moves(self):
x, y = self.current_position
# Define possible moves: up, down, left, right
moves = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
return moves
def move_robot(self):
# Loop until robot reaches the goal
while self.current_position != self.goal:
possible_moves = self.get_possible_moves()
# Select the move with the minimum heuristic distance to the goal
next_move = min(possible_moves, key=self.heuristic_distance)
# Update the robot's current position
self.current_position = next_move
print(f'Moving to {self.current_position}, Distance to goal:
{self.heuristic_distance(next_move)}')
# Define start and goal positions
start_position = (0, 0)
goal_position = (3, 3)
# Initialize the robot traversal
robot = RobotTraversal(start_position, goal_position)
robot.move_robot()
```

Output:

```
Moving to (1, 0), Distance to goal: 5
Moving to (2, 0), Distance to goal: 4
Moving to (3, 0), Distance to goal: 3
Moving to (3, 1), Distance to goal: 2
Moving to (3, 2), Distance to goal: 1
Moving to (3, 3), Distance to goal: 0
```

Result:

thus program is to solve a robot traversal problem on a grid using Means-End Analysis is executed successfully and verified.

Travelling Salesman Problem

Aim:

The aim of this program is to solve the Traveling Salesman Problem (TSP) using a naive (brute-force) approach in Python.

Alogrithm:

1. Define Parameters:

The graph representing distances between cities is given as an adjacency matrix.

s: The starting city (in this case, city 0).

2. Generate All Possible Routes:

Create a list vertex of all cities except the starting city s.

Use Python's `itertools.permutations` to generate all permutations of these cities, representing all possible routes the salesman can take.

3. Calculate Path Weights:

For each permutation (possible route) of cities:

Initialize `current_pathweight` to store the cost of the current route.

Iterate through the cities in the current route, summing up the distances from one city to the next in `current_pathweight`.

Finally, add the distance from the last city in the route back to the starting city to complete the cycle.

4. Find the Minimum Path:

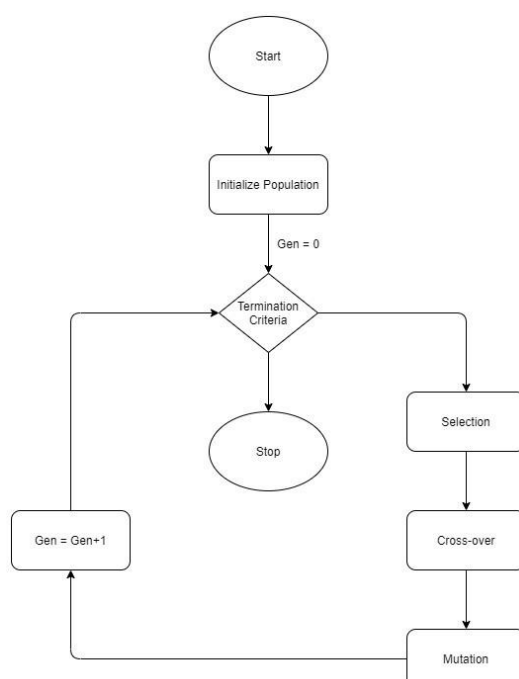
Keep track of the minimum path cost (`min_path`) by comparing `current_pathweight` of each route with the current `min_path` value.

Update `min_path` whenever a smaller path cost is found.

5. Return the Result:

After evaluating all routes, return the minimum path cost found.

Flowchart:



Program:

```
# 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
    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:

```
p/mutli agent/trave  
80
```

Result:

Thus program is to solve the Traveling Salesman Problem (TSP) using a naive(brute-force) approach in Python is executed successfully and verified.