Practical: 1

Aim: Write a program to implement Tic-Tac-Toe game problem.

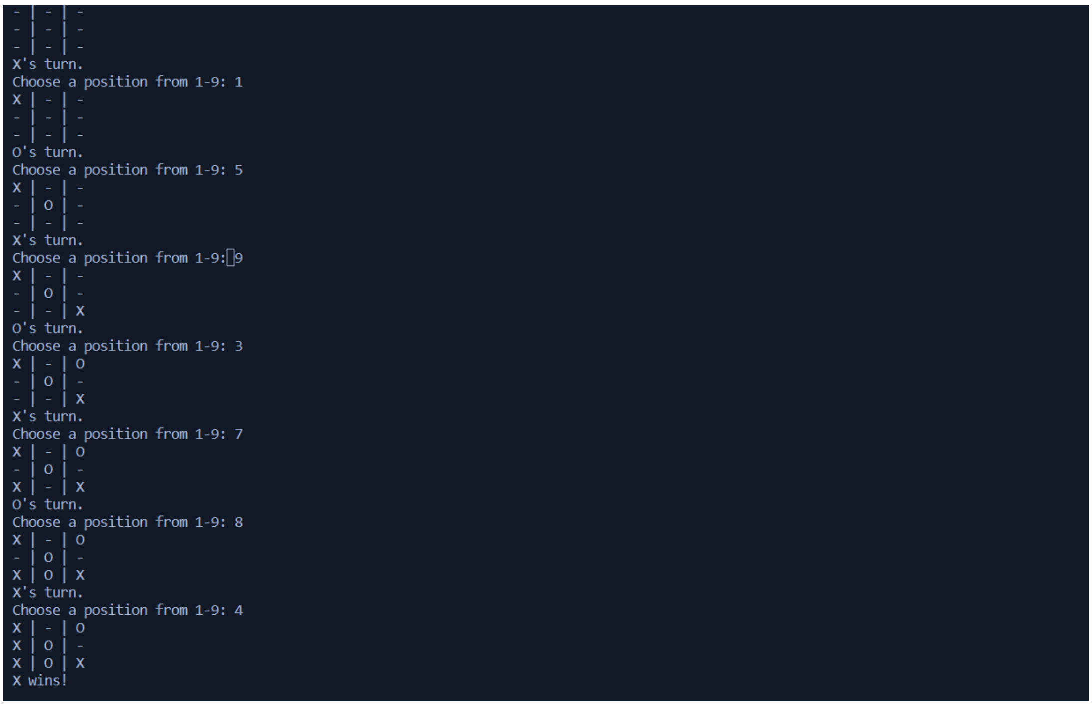
Code:

```
board =         ["-", "-", "-",
                "-", "-", "-",
                "-", "-", "-"]

def print_board():
        print(board[0] + " | " + board[1] + " | " + board[2])
        print(board[3] + " | " + board[4] + " | " + board[5])
        print(board[6] + " | " + board[7] + " | " + board[8])

def take_turn(player):
        print(player + "'s turn.")
        position = input("Choose a position from 1-9: ")
        while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
                position = input("Invalid input. Choose a position from 1-9: ")
        position = int(position) - 1
        while board[position] != "-":
                position = int(input("Position already taken. Choose a different position: ")) -
1
        board[position] = player
        print_board()

def check_game_over():
        # Check for a win
        if (board[0] == board[1] == board[2] != "-") or \
        (board[3] == board[4] == board[5] != "-") or \
        (board[6] == board[7] == board[8] != "-") or \
        (board[0] == board[3] == board[6] != "-") or \
        (board[1] == board[4] == board[7] != "-") or \
        (board[2] == board[5] == board[8] != "-") or \
        (board[0] == board[4] == board[8] != "-") or \
        (board[2] == board[4] == board[6] != "-"):
                return "win"
        elif "-" not in board:
                return "tie"
        else:
                return "play"
def play_game():
        print_board()
        current_player = "X"
        game_over = False
        while not game_over:
                take_turn(current_player)
                game_result = check_game_over()
```

```
        if game_result == "win":
            print(current_player + " wins!")
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            # Switch to the other player
            current_player = "O" if current_player == "X" else "X"

play_game()
```

Output:

Case 1:



```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 1
X | - | -
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 5
X | - | -
- | O | -
- | - | -
X's turn.
Choose a position from 1-9: 9
X | - | -
- | O | -
- | - | X
O's turn.
Choose a position from 1-9: 3
X | - | O
- | O | -
- | - | X
X's turn.
Choose a position from 1-9: 7
X | - | O
- | O | -
X | - | X
O's turn.
Choose a position from 1-9: 8
X | - | O
- | O | -
X | O | X
X's turn.
Choose a position from 1-9: 4
X | - | O
X | O | -
X | O | X
X wins!
```

Case 2:

```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 1
X | - | -
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 3
X | - | O
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 2
X | X | O
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 6
X | X | O
- | - | O
- | - | -
X's turn.
Choose a position from 1-9: 4
X | X | O
X | - | O
- | - | -
O's turn.
Choose a position from 1-9: 9
X | X | O
X | - | O
- | - | O
O wins!
```

Case 3:

```
Choose a position from 1-9: 2
X | O | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 3
X | O | X
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 7
X | O | X
- | - | -
O | - | -
X's turn.
Choose a position from 1-9: 8
X | O | X
- | - | -
O | X | -
O's turn.
Choose a position from 1-9: 9
X | O | X
- | - | -
O | X | O
X's turn.
Choose a position from 1-9: 4
X | O | X
X | - | -
O | X | O
O's turn.
Choose a position from 1-9: 5
X | O | X
X | O | -
O | X | O
X's turn.
Choose a position from 1-9: 6
X | O | X
X | O | X
O | X | O
It's a tie!
```

Practical:02

Aim: - Write a program to implement BFS (for 8 puzzle problem or Water Jugproblem or any AI search problem).

Code:

```
from collections import deque

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

def print_state(state):
    for row in state:
        print(row)

def possible_moves(state):
    moves = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                if i > 0:
                    moves.append((i, j, i - 1, j))
                if i < 2:
                    moves.append((i, j, i + 1, j))
                if j > 0:
                    moves.append((i, j, i, j - 1))
                if j < 2:
                    moves.append((i, j, i, j + 1))
    return moves

def bfs(initial_state, goal_state):
    visited = set()
    queue = deque()
    queue.append((initial_state, []))

    while queue:
        current_state, path = queue.popleft()
        visited.add(tuple(map(tuple, current_state)))

        if current_state == goal_state:
            print("Goal state found!")
            print("Path to the goal:")
            for step in path:
                print_state(step)
                print()
            return

        for move in possible_moves(current_state):
```

```
            i, j, new_i, new_j = move
            new_state = [list(row) for row in current_state]
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[i][j]

            if tuple(map(tuple, new_state)) not in visited:
                queue.append((new_state, path + [new_state]))

    print("Goal state is not reachable.")

bfs(initial_state, goal_state)
```

Output:

```
Goal state found!
Path to the goal:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Practical:3

Aim: Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

Code:

```
def water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1=0, jug2=0,
   visited=set()): if jug1 == target_amount and jug2 == 0:
     return [(jug1,
jug2)]
visited.add((jug1,
jug2))
if jug1 < jug1_capacity and (jug1_capacity, jug2) not in visited:
  path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1_capacity,
jug2, visited) if path:
   return [(jug1, jug2)] + path
if jug2 < jug2_capacity and (jug1, jug2_capacity) not in visited:
  path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1,
jug2_capacity, visited) if path:
   return [(jug1, jug2)] +
path if path:
   return [(jug1, jug2)] + path
if jug2 > 0 and (jug1, 0) not in visited:
   path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1, 0,
visited) if path:
     return [(jug1, jug2)] + path
if jug1 > 0 and jug2 < jug2_capacity:
pour_amount = min(jug1, jug2_capacity - jug2)
path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1 - pour_amount,
jug2 + pour_amount, visited)
```

```
if path:
  return [(jug1, jug2)] + path
pour_amount = min(jug2, jug1_capacity - jug1)
path = water_jug_dfs(jug1_capacity, jug2_capacity, target_amount, jug1 + pour_amount,
jug2 - pour_amount, visited)
   if path:
    return [(jug1, jug2)] + path return []
jug1_capacity = 4 jug2_capacity= 3 target_amount = 2
solution = water_jug_dfs(jug1_capacity, jug2_capacity,
target_amount) if solution:
    for step, state
    inenumerate(solution):
    print(f"Step{p+1}: {state}")
else:
    print("No solution found.")
```

Output:

```
Step 1: (0, 0)
Step 2: (4, 0)
Step 3: (1, 3)
Step 4: (1, 0)
Step 5: (0, 1)
Step 6: (4, 1)
Step 7: (2, 3)
Step 8: (2, 0)
```

Practical: 4

Aim: - Write a program to implement Single Player Game (Using any Heuristic Function)

Code: -

```
import random

def heuristic_guess(low, high):
    return (low + high) // 2

def play_game():
    print("Welcome to the Guessing Game!")
    target_number = random.randint(1, 100)
    low, high = 1, 100
    attempts = 0

    while True:
        guess = heuristic_guess(low, high)
        print(f"I guess {guess}")

        if guess == target_number:
            print(f"Congratulations! I guessed the number {target_number} in {attempts} attempts.")
            break
        elif guess < target_number:
            print("Too low!")
            low = guess + 1
        else:
            print("Too high!")
            high = guess - 1

        attempts += 1

if__name__== "_main_":
    play_game()
```

Output:

```
Welcome to the Guessing Game!
I guess 50
Too low!
I guess 75
Too low!
I guess 88
Too low!
I guess 94
Too low!
I guess 97
Too high!
I guess 95
Too low!
I guess 96
Congratulations! I guessed the number 96 in 6 attempts.
```

Practical: 5

AIM: Implement A* algorithm.

Code:

```
import heap

class Node:
    def__init_(self, state, parent=None, action=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic

    def__lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start_state, goal_state, get_neighbors, heuristic):
    open_list = []
    closed_set = set()
    start_node = Node(start_state, None, None, 0, heuristic(start_state, goal_state))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.state == goal_state:
            path = []
            while current_node:
                path.append((current_node.state, current_node.action))
                current_node = current_node.parent
            return list(reversed(path))

        closed_set.add(current_node.state)
        for neighbor_state, action, step_cost in get_neighbors(current_node.state):
            if neighbor_state in closed_set:
                continue
            g_score = current_node.cost + step_cost
            h_score = heuristic(neighbor_state, goal_state)
            f_score = g_score + h_score
            neighbor_node = Node(neighbor_state, current_node, action, g_score, h_score)

            # Check if the neighbor is already in the open list
            found = False
            for node in open_list:
                if node.state == neighbor_state:
                    found = True
                    if g_score < node.cost:
                        open_list.remove(node)
                        heapq.heappush(open_list, neighbor_node)
```

```
                    break
            if not found:
                heapq.heappush(open_list, neighbor_node)

    return None  # No path found

def get_neighbors(state):
    x, y = state
    neighbors = []
    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < len(grid) and 0 <= new_y < len(grid[0]) and grid[new_x][new_y] == 0:
            neighbors.append(((new_x, new_y), f"Move to ({new_x}, {new_y})", 1))
    return neighbors

def manhattan_distance(state, goal):
    x1, y1 = state
    x2, y2 = goal
    return abs(x1 - x2) + abs(y1 - y2)

# Example usage
start = (0, 0)
goal = (3, 3)
grid = [[0, 0, 0, 0], [1, 1, 0, 1], [0, 0, 0, 0], [1, 0, 1, 0]]

path = astar(start, goal, get_neighbors, manhattan_distance)

if path:
    for state, action in path:
        print(f"Action: {action}, State: {state}")
else:
    print("No path found.")
```

Output:

```
Action: None, State: (0, 0)
Action: Move to (0, 1), State: (0, 1)
Action: Move to (0, 2), State: (0, 2)
Action: Move to (1, 2), State: (1, 2)
Action: Move to (2, 2), State: (2, 2)
Action: Move to (2, 3), State: (2, 3)
Action: Move to (3, 3), State: (3, 3)
```

Practical: 6

Aim: Write a program to implement mini-max algorithm for any game development.

Code:

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if curDepth == targetDepth:
        return scores[nodeIndex]

    if maxTurn:
        return max(
            minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
            minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth)
        )
    else:
        return min(
            minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
            minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth)
        )

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = int(math.log2(len(scores)))

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

Output:

```
The optimal value is : 12
```

Practical: 7

Aim: Assume given a set of facts of the form father (name1, name2) (name1is the father of name2).

Code:

```
female(pam).
female(liz).
female(pat).
female(ann).

male(jim).
male(bob).
male(tom).
male(peter).

parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).

parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).

mother(X,Y):- parent(X,Y),female(X).
father(X,Y):-parent(X,Y),male(X).
sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
wife(X,Y):-
parent(X,Z),parent(Y,Z),female(X),male(Y).
uncle(X,Z):-brother(X,Y),parent(Y,Z).
```

Output:

Practical: 8

Aim : Define a predicate brother(X,Y) which holds iff X and Y arebrothers.
Define a predicate cousin(X,Y) which holds iff X and Y are cousins.
Define a predicate grandson(X,Y) which holds iff X is a grandson of Y.
Define a predicate descendent(X,Y) which holds iff X is a descendent of Y.
Consider the following genealogical tree: father(a,b). father(a,c). father(b,d). father(b,e). father(c,f).
Say which answers, and in which order, are generated by your definitions for the following queries in Prolog:
?- brother(X,Y).
?- cousin(X,Y).
?- grandson(X,Y).
?- descendent(X,Y).

Code:

```
father(kevin,milu).

father(kevin,yash).

father(milu,meet).

father(milu,raj).

father(yash,jay).



brother(X,Y):-father(K,X),father(K,Y).

cousin(A,B):-father(K,X),father(K,Y),father(X,A),father(Y,B).

grandson(X,Y):-father(X,K),father(K,Y).

descendent(X,Y):-father(K,X),father(K,Y).

descendent(X,Y):-father(K,X),father(K,Y),father(X,A),father(Y,B).
```

Output:

```
brother(X,Y).
Singleton variables: [A,B]
X = Y, Y = milu
X = milu,
Y = yash
X = yash,
Y = milu
X = Y, Y = yash
```

Practical:9

Aim: Write a program to solve Tower of Hanoi problem using Prolog.

Code:

```
move(1,X,Y,_) :-write('Move top disk from '), write(X), write(' to '),write(Y), nl.
move(N,X,Y,Z) :-
N>1,
M is N-1, move(M,X,Z,Y),
move(1,X,Y,_),
move(M,Z,Y,X).
```

Output:

```
move(1,X,Y,Z) .
Move top disk from _6768 to _6776
true
move(3,X,Y,Z) .
Move top disk from _6768 to _6776
Move top disk from _6768 to _6772
Move top disk from _6776 to _500
Move top disk from _496 to _504
Move top disk from _500 to _496
Move top disk from _500 to _504
Move top disk from _496 to _504
true
```

Practical:10

Aim: Write a program to solve N-Queens problem using Prolog.

Code:

```
% render solutions nicely.
:- use_rendering(chess).

%%    queens(+N, -Queens) is nondet.
%
%       @param          Queens is a list of column numbers for placing the queens.
%       @author Richard A. O'Keefe (The Craft of Prolog)

queens(N, Queens) :-
    length(Queens, N),
        board(Queens, Board, 0, N, _, _),
        queens(Board, 0, Queens).

board([], [], N, N, _, _).
board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-
        Col is Col0+1,
        functor(Vars, f, N),
        constraints(N, Vars, VR, VC),
        board(Queens, Board, Col, N, VR, [_|VC]).

constraints(0, _, _, _) :- !.
constraints(N, Row, [R|Rs], [C|Cs]) :-
        arg(N, Row, R-C),
        M is N-1,
        constraints(M, Row, Rs, Cs).

queens([], _, []).
queens([C|Cs], Row0, [Col|Solution]) :-
        Row is Row0+1,
        select(Col-Vars, [C|Cs], Board),
        arg(Row, Vars, Row-Row),
        queens(Board, Row, Solution).
```

Output:

Practical:11

Aim: Write a program to solve 8 puzzle problem using Prolog.

Code:

```
goal([1,2,3, 4,0,5, 6,7,8]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
[0,X1,X3, X4,X5,X6, X7,X8,X9]).
move([X1,X2,0, X4,X5,X6, X7,X8,X9],
[X1,0,X2, X4,X5,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6,X7,X8,X9],
[X1,X2,X3, 0,X4,X6,X7,X8,X9]).
move([X1,X2,X3, X4,X5,0,X7,X8,X9],
[X1,X2,X3, X4,0,X5,X7,X8,X9]).
move([X1,X2,X3, X4,X5,X6, X7,0,X9],
[X1,X2,X3, X4,X5,X6, 0,X7,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
[X1,X2,X3, X4,X5,X6, X7,0,X8]).
move([0,X2,X3, X4,X5,X6, X7,X8,X9],
[X2,0,X3, X4,X5,X6, X7,X8,X9]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
[X1,X3,0, X4,X5,X6, X7,X8,X9]).
move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
[X1,X2,X3, X5,0,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
[X1,X2,X3, X4,X6,0, X7,X8,X9]).
move([X1,X2,X3, X4,X5,X6,0,X8,X9],
[X1,X2,X3, X4,X5,X6,X8,0,X9]).
move([X1,X2,X3, X4,X5,X6,X7,0,X9],
[X1,X2,X3, X4,X5,X6,X7,X9,0]).
move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
[0,X2,X3, X1,X5,X6, X7,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
[X1,0,X3, X4,X2,X6, X7,X8,X9]).
move([X1,X2,X3, X4,X5,0, X7,X8,X9],
[X1,X2,0, X4,X5,X3, X7,X8,X9]).
move([X1,X2,X3, X4,X5,X6, X7,0,X9],
[X1,X2,X3, X4,0,X6, X7,X5,X9]).
move([X1,X2,X3, X4,X5,X6, X7,X8,0],
[X1,X2,X3, X4,X5,0, X7,X8,X6]).
move([X1,X2,X3, X4,X5,X6, 0,X8,X9],
[X1,X2,X3, 0,X5,X6, X4,X8,X9]).
move([0,X2,X3, X4,X5,X6, X7,X8,X9],
[X4,X2,X3, 0,X5,X6, X7,X8,X9]).
move([X1,0,X3, X4,X5,X6, X7,X8,X9],
[X1,X5,X3, X4,0,X6, X7,X8,X9]).

move([X1,X2,0, X4,X5,X6, X7,X8,X9],
[X1,X2,X6, X4,X5,0, X7,X8,X9]).
move([X1,X2,X3, 0,X5,X6, X7,X8,X9],
```

```
[X1,X2,X3, X7,X5,X6, 0,X8,X9]).
move([X1,X2,X3, X4,0,X6, X7,X8,X9],
[X1,X2,X3, X4,X8,X6, X7,0,X9]).
move([X1,X2,X3, X4,X5,0, X7,X8,X9],
[X1,X2,X3, X4,X5,X9, X7,X8,0]).
dfsSimplest(S, [S]) :- goal(S).
dfsSimplest(S, [S|Rest]) :- move(S, S2), dfsSimplest(S2, Rest).
dfs(S, Path, Path) :- goal(S).
dfs(S, Checked, Path) :- move(S, S2), \+member(S2, Checked), dfs(S2, [S2|Checked], Path).
```

Output:

```
dfsSimplest(S, [S]).
S = [1, 2, 3, 4, 0, 5, 6, 7, 8]
dfsSimplest(S, [S|Rest]).
Rest = [],
S = [1, 2, 3, 4, 0, 5, 6, 7, 8]
dfs(S, Path, Path).
S = [1, 2, 3, 4, 0, 5, 6, 7, 8]
dfs(S, Checked, Path).
Checked = Path,
S = [1, 2, 3, 4, 0, 5, 6, 7, 8]
```

Practical:12

Aim: Write a program to solve travelling salesman problem using Prolog.

Code:

```
edge(a, b, 3).
edge(a, c, 4).
edge(a, d, 2).
edge(a, e, 1).
edge(b, c, 4).
edge(b, d, 6).
edge(b, e, 3).
edge(c, d, 5).
edge(c, e, 2).
edge(d, e, 6).
edge(b, a, 3).
edge(c, a, 4).
edge(d, a, 2).
edge(e, a, 6).
edge(c, b, 4).
edge(d, b, 5).
edge(e, b, 3).
edge(d, c, 5).
edge(e, c, 2).
edge(e, d, 6).
edge(a, h, 2).
edge(h, d, 1).
len([], 0).
len([H|T], N):- len(T, X), N is X+1 .
best_path(Visited, Total):- path(a, a, Visited, Total).
path(Start, Fin, Visited, Total) :- path(Start, Fin, [Start], Visited, 0, Total).
path(Start, Fin, CurrentLoc, Visited, Costn, Total) :- edge(Start, StopLoc, Distance),
NewCostn is Costn + Distance, \+ member(StopLoc,CurrentLoc),
path(StopLoc, Fin, [StopLoc|CurrentLoc], Visited, NewCostn, Total).
path(Start, Fin, CurrentLoc, Visited, Costn, Total) :- edge(Start, Fin, Distance),
reverse([Fin|CurrentLoc], Visited), len(Visited, Q), (Q\=7 -> Total is 100000; Total is Costn
+ Distance).
shortest_path(Path):- setof(Cost-Path, best_path(Path,Cost), Holder),pick(Holder,Path).
best(Cost-Holder,Bcost-_,Cost-Holder):- Cost<Bcost,!.
best(_,X,X).
pick([Cost-Holder|R],X):- pick(R,Bcost-Bholder),best(Cost-Holder,Bcost-Bholder,X),!.
pick([X],X).
```

Output:

```
shortest_path(Path).
Singleton variables: [H]
Path = 16-[a, h, d, c, e, b, a]
```