

## **EXPERIMENT NO – 01**

**AIM:** Design/construct the workflow of a general AI project using draw.io

**OBJECTIVE:** To design/construct the workflow of a general AI project using draw.io

### **DESCRIPTION:**

Here, we are taking up three AI Projects. They are:

- Automated Taxi Driver
- Automated English Tutor
- Automated Diagnosis System

#### **Automated Taxi Driver:**

Consider the task of designing an automated Taxi Driver:

- Agent: Automated Taxi Driver
- Performance Measure: Safe, fast, legal, comfortable trip, maximize profits
- Environment: Roads, other traffic, pedestrians, customers
- Actuators: Steering wheel, accelerator, brake, signal, horn
- Sensors: Cameras, sonar, speedometer, GPS, engine sensors, keyboard

#### **Automated English Tutor:**

Consider the task of designing an automated English Tutor:

- Agent: Automated English Tutor
- Performance measure: Maximize student's score on test
- Environment: Set of students
- Actuators: Screen display (Exercises, suggestions, corrections)
- Sensors: Keyboard

#### **Automated Diagnosis System:**

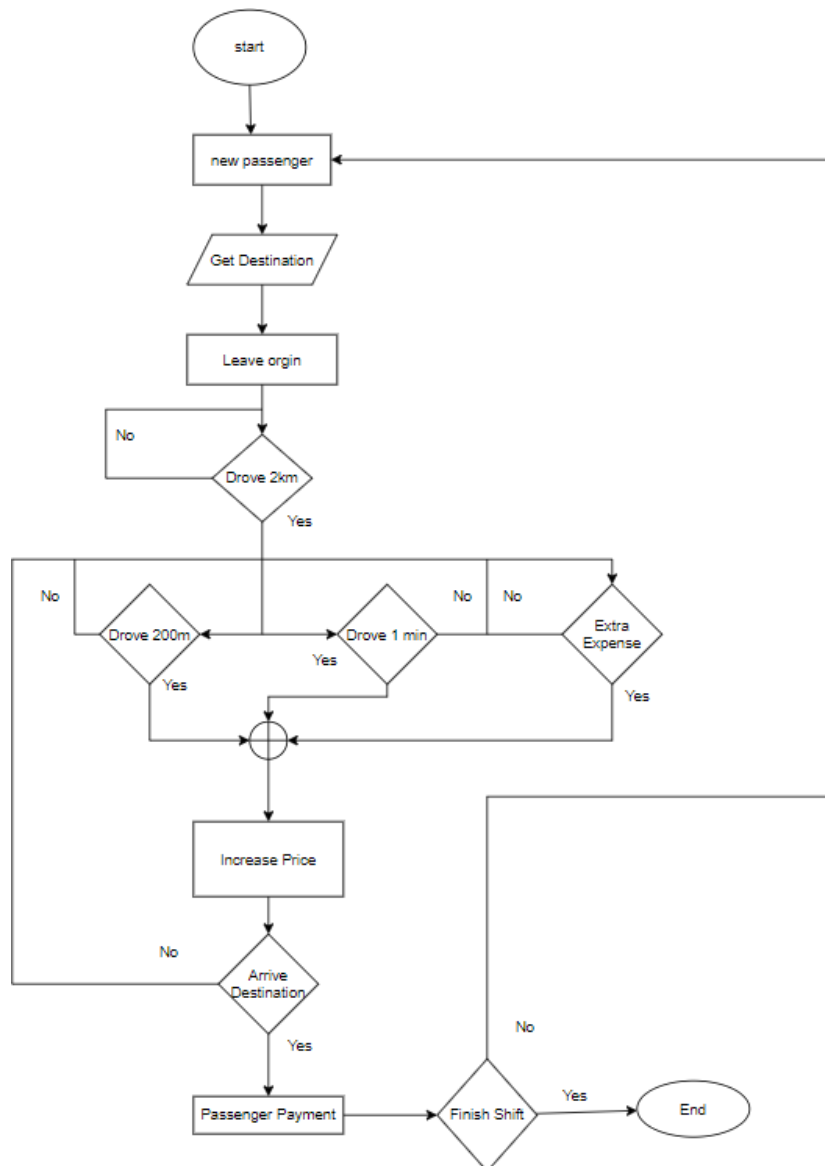
Consider the task of designing an automated Diagnosis System:

- Agent: Medical diagnosis system

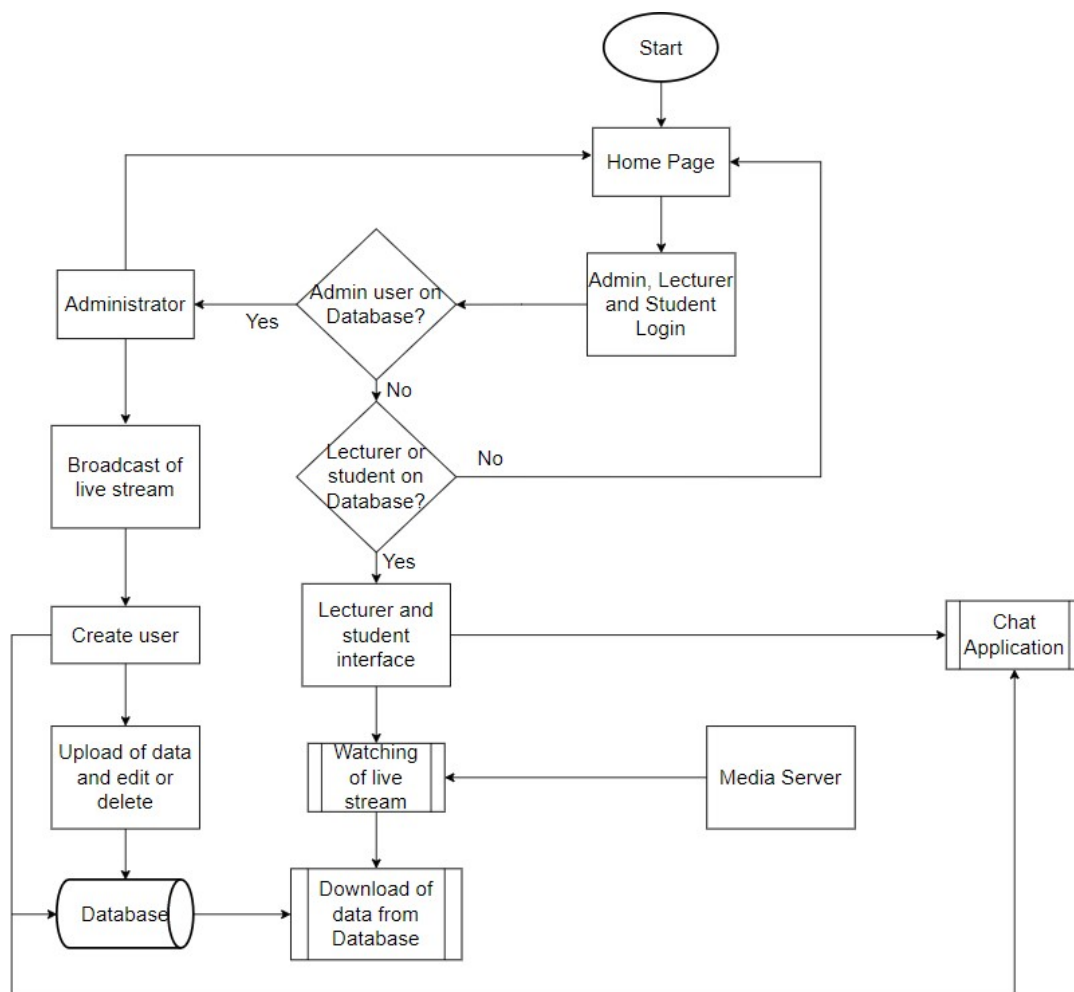
- Performance: Healthy patient, minimize costs, lawsuits
- Environment: Patient, hospital, staff
- Actuators: Screen display (questions, tests, diagnosis, treatments, referrals)
- Sensors: Keyboard (entry of symptoms, finding, patient's answers)

## FLOW CHARTS:

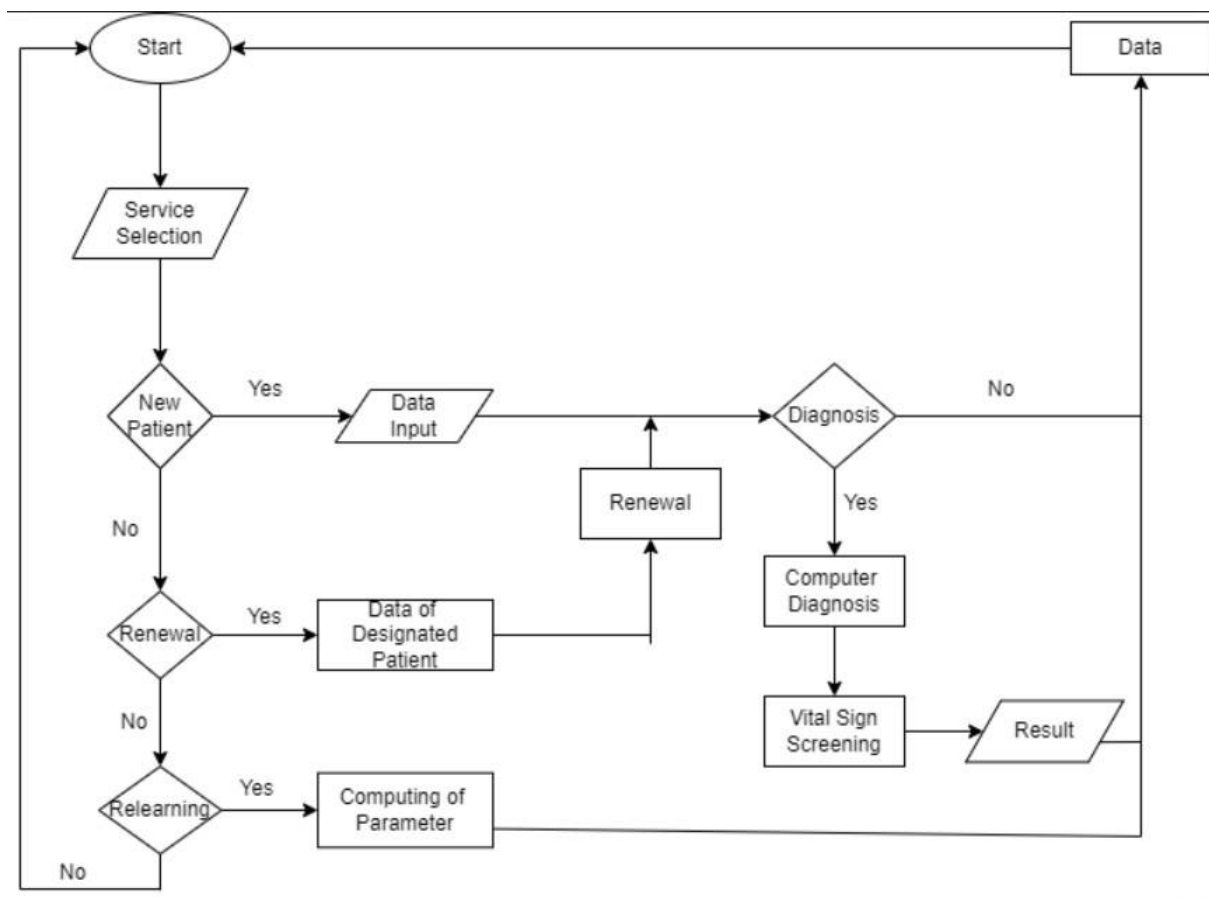
### Automated Taxi Driver:



## Automated English Tutor:



## Automated Diagnosis System:



## CONCLUSION:

We have successfully designed/constructed the workflow of a general AI project using draw.io

## EXPERIMENT NO – 02

**AIM:** Implement Water Jug Problem

**OBJECTIVE:** To implement Water Jug Problem using BFS, DFS, and Memoization

### DESCRIPTION:

Water Jug Problem can be defined as You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring mark on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug.

**BFS:** Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

**DFS:** It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

**Memoization:** In computing, memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.

### ALGORITHM:

Step-1: Start

Step-2: Fill the 3-litre jug completely.

Step-3: Transfer 3 litres from a 3-liter jug to a 4-liter jug.

Step-4: Now, fill the 3-liter jug fully.

Step-5: Pour 1 litre from a 3-liter jug into a 4-liter jug.

Step-6: Now, in 3-litre jug 2-litre water will be left

Step-7: Empty the 4-litre Jug

Step-8: Pour 2-litre from a 3-litre jug into a 4-litre jug

Step-9: Therefore, in a 4-litre jug 2-litre water will be present. Hence, Goal State reached

Step-10: End

**PROGRAM AND OUTPUT:**

**WaterJugUsingBFS.py:**

```
from collections import deque
```

```

def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))
    while (len(q) > 0):
        u = q.popleft()
        if ((u[0], u[1]) in m):
            continue
        if ((u[0] > a or u[1] > b or
            u[0] < 0 or u[1] < 0)):
            continue
        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1
        if (u[0] == target or u[1] == target):
            isSolvable = True
            if (u[0] == target):
                if (u[1] != 0):
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):
                    path.append([0, u[1]])
        sz = len(path)
        for i in range(sz):
            print("(", path[i][0], ",",
                path[i][1], ")")
        break
    q.append([u[0], b])
    q.append([a, u[1]])
    for ap in range(max(a, b) + 1):
        c = u[0] + ap
        d = u[1] - ap
        if (c == a or (d == 0 and d >= 0)):
            q.append([c, d])

```



```
c = u[0] - ap
d = u[1] + ap
if ((c == 0 and c >= 0) or d == b):
    q.append([c, d])
```

```

        q.append([a, 0])
        q.append([0, b])
    if (not isSolvable):
        print("Solution not possible")

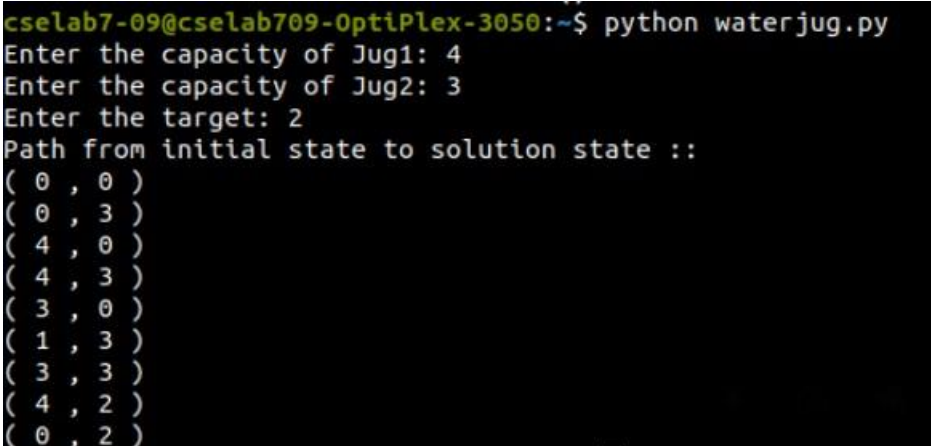
if __name__ == '__main__':

    Jug1 = int(input("Enter the capacity of Jug1: "))
    Jug2 = int(input("Enter the capacity of Jug2: "))
    target = int(input("Enter the target: "))
    print("Path from initial state "
          "to solution state ::")

    Solution(Jug1, Jug2, target)

```

### Output:



```

cse1ab7-09@cse1ab709-OptiPlex-3050:~$ python waterjug.py
Enter the capacity of Jug1: 4
Enter the capacity of Jug2: 3
Enter the target: 2
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )

```

### WaterJugUsingDFS.py:

```
class Node:
```

```

    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

```

```

    def get_child_nodes(self, capacities):
        a, b = self.state
        max_a, max_b = capacities

```

```
children = []
```

```

        children.append(Node((max_a, b), self))
        children.append(Node((a, max_b), self))
        children.append(Node((0, b), self))
        children.append(Node((a, 0), self))
        if a + b >= max_b:
            children.append(Node((a - (max_b - b), max_b), self))
        else:
            children.append(Node((0, a + b), self))
        if a + b >= max_a:
            children.append(Node((max_a, b - (max_a - a)), self))
        else:
            children.append(Node((a + b, 0), self))
        return children
def dfs(start_state, goal_state, capacities):
    start_node = Node(start_state, None)
    visited = set()
    stack = [start_node]

    while stack:
        node = stack.pop()
        if node.state == goal_state:
            path = []
            while node.parent:
                path.append(node.state)
                node = node.parent
            path.append(start_state)
            path.reverse()
            return path
        if node.state not in visited:
            visited.add(node.state)
            for child in node.get_child_nodes(capacities):
                stack.append(child)
    return None
start_state = (0, 0)
a,b=map(int, input("Enter the capacities of jugs: ").split())
c,d=map(int, input("Enter the capacities of goal state: ").split())
goal_state = (c, d)
capacities = (a, b)

path = dfs(start_state, goal_state, capacities);print(path)

```

### Output:

```
cseLab7-09@cseLab709-OptiPlex-3050:~$ python waterjug1.py
Enter the capacities of jugs: 4 3
Enter the capacities of goal state: 2 0
[(0, 0), (0, 3), (3, 0), (3, 3), (4, 2), (4, 0), (1, 3), (1, 0), (0, 1), (4, 1), (2, 3), (2, 0)]
```

### WaterJugUsingMemoization.py:

```
from collections import defaultdict
jug1, jug2, aim = 4, 3, 2
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                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))))
    else:
        return False
print("Steps: ")
waterJugSolver(0, 0)
```

### Output:

```
cseLab7-09@cseLab709-OptiPlex-3050:~$ python waterjug2.py
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

### CONCLUSION:

We have successfully implemented Water Jug Problem using BFS, DFS, and Memoization.

## **EXPERIMENT NO – 03**

**AIM:** Implement search problem using A\* Search.

**OBJECTIVE:** To implement a search problem using the A\* Algorithm.

### **DESCRIPTION:**

The A\* algorithm is a search algorithm that is widely used in artificial intelligence and game development. It combines the use of a heuristic function and a cost function to estimate the cost of reaching the goal state and measure the actual cost of the path taken to reach the current state. The algorithm is useful for finding the shortest path between two points in a graph, and it updates the cost of each node in the graph based on the cost function and the heuristic function. The algorithm continues to expand nodes until it reaches the goal state or determines that there is no path to the goal state. The A\* algorithm is known for its efficiency and effectiveness in finding the shortest path and has been used in various real-world applications.

### **ALGORITHM:**

1. Initialize the start node with a cost of zero and add it to the open list.
2. While the open list is not empty, do the following:
  - Select the node with the lowest cost from the open list.
  - If the selected node is the goal state, terminate the algorithm and return the path to the goal.
  - Expand the selected node by generating its successor nodes.
  - For each successor node, calculate the cost function and the heuristic function, and add it to the open list.
  - If a successor node is already in the open list or the closed list, update its cost if a better path has been found. f. Add the selected node to the closed list.
3. If the open list is empty and the goal state has not been reached, terminate the algorithm and return failure.

### **PROGRAM:**

```
def astaralgo(start_node, stop_node):  
    open_set = set(start_node)
```

```
closed_set=set()
```

```

parents={}
g={}

g[start_node]=0
parents[start_node]=start_node
while len(open_set)>0:
    #print("***")
    n=None
    for v in open_set:
        if n==None or g[v]+heuristic(v)<g[n]+heuristic(n):
            n=v
    if n==stop_node or Graph_nodes[n]==None:
        pass
    else:
        for (m,w) in get_neighbors(n):
            if m not in open_set and m not in closed_set:
                parents[m]=n
                open_set.add(m)

                g[m]=g[n]+w
            else:
                if g[m]>g[n]+w:
                    g[m]=g[n]+w
                    parents[m]=n
                    if m in closed_set:
                        closed_set.remove(m)
                    open_set.add(m)
        if n==None:
            print("Path doesn't exist")
            return None
        if n==stop_node:
            path=[]
            while parents[n]!=n:
                path.append(n)
                #print("&&")
                n=parents[n]
            n=parents[n]
            path.append(start_node)
            path.reverse()

```



```
print(f"path from {start_node} to {stop_node} is : ")
```

```

        print(path)
        return path
    open_set.remove(n)
    closed_set.add(n)
    print("NO path exists")
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
astaralgo('A', 'G')

```

### OUTPUT:

```

path from A to G is :
['A', 'E', 'D', 'G']

```

### CONCLUSION:

We have successfully implemented a search problem using the A\* search

technique.

## EXPERIMENT NO – 04

**AIM:** Implement an 8-puzzle problem solver using Heuristic search technique

**OBJECTIVE:** To implement an 8-puzzle problem solver using Heuristic search technique

### DESCRIPTION:

8-puzzle is a variant of the 15-puzzle. You can check it out at [https://en.wikipedia.org/wiki/15\\_puzzle](https://en.wikipedia.org/wiki/15_puzzle). You will be presented with a randomized grid and your goal is to get it back to the original ordered configuration. You can play the game to get familiar with it at <http://mypuzzle.org/sliding>. We will use an A\* algorithm to solve this problem. It is an algorithm that's used to find paths to the solution in a graph. This algorithm is a combination of Dijkstra's algorithm and a greedy best-first search. Instead of blindly guessing where to go next, the A\* algorithm picks the one that looks the most promising. At each node, we generate the list of all possibilities and then pick the one with the minimal cost required to reach the goal. At each node, we need to compute the cost. This cost is basically the sum of two costs – the first cost is the cost of getting to the current node and the second cost is the cost of reaching the goal from the current node. We use this summation as our heuristic.

### ALGORITHM:

Step-1: Start

Step-2: Define open and closed list

Step-3: First move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state.

Step-4: After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list.

Step-5: A state with the least f-score is selected and expanded again.

Step-6: This process continues until the goal state occurs as the current state.

Step-7: End

### PROGRAM:

```
from simpleai.search import astar, SearchProblem
class PuzzleSolver(SearchProblem):
```

```
def actions(self, cur_state):  
    rows = string_to_list(cur_state)
```

```

row_empty, col_empty = get_location(rows, 'e')
actions = []
if row_empty > 0:
    actions.append(rows[row_empty - 1][col_empty])
if row_empty < 2:
    actions.append(rows[row_empty + 1][col_empty])
if col_empty > 0:
    actions.append(rows[row_empty][col_empty - 1])
if col_empty < 2:
    actions.append(rows[row_empty][col_empty + 1])
return actions
def result(self, state, action):
    rows =
    string_to_list(state)
    row_empty, col_empty = get_location(rows, 'e')
    row_new, col_new = get_location(rows, action)
    rows[row_empty][col_empty], rows[row_new][col_new] = \
        rows[row_new][col_new], rows[row_empty][col_empty]
    return list_to_string(rows)
def is_goal(self, state):
    return state == GOAL
def heuristic(self, state):
    rows =
    string_to_list(state)
    distance = 0
    for number in '12345678e':
        row_new, col_new = get_location(rows, number)
        row_new_goal, col_new_goal = goal_positions[number]
        distance += abs(row_new - row_new_goal) + abs(col_new -
col_new_goal)
    return distance
def list_to_string(input_list):
    return '\n'.join(['-'.join(x) for x in input_list])
def string_to_list(input_string):
    return [x.split('-') for x in input_string.split('\n')]
def get_location(rows, input_element):
    for i, row in enumerate(rows):
        for j, item in enumerate(row):
            if item == input_element:

```

```
GOAL = ""1-2-3
4-5-6
return i, j
```

```

7-8-e'''
INITIAL = '''1-e-2
6-3-4
7-5-8'''

goal_positions = {}

rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = get_location(rows_goal, number)
result = astar(PuzzleSolver(INITIAL))
for i, (action, state) in enumerate(result.path()):
    print()
    if action == None:
        print('Initial configuration')
    elif i == len(result.path()) - 1:
        print('After moving', action, 'into the empty space. Goal achieved!')
    else:
        print('After moving', action, 'into the empty space')
    print(state)

```



## OUTPUT:

```
cse1ab7-09@cse1ab709-OptiPlex-3050:~$ python 8puzzle.py
```

```
Initial configuration
```

```
1-e-2  
6-3-4  
7-5-8
```

```
After moving 2 into the empty space
```

```
1-2-e  
6-3-4  
7-5-8
```

```
After moving 4 into the empty space
```

```
1-2-4  
6-3-e  
7-5-8
```

```
After moving 3 into the empty space
```

```
1-2-4  
6-e-3  
7-5-8
```

```
After moving 6 into the empty space
```

```
1-2-4  
e-6-3  
7-5-8
```

```
After moving 1 into the empty space
```

```
e-2-4  
1-6-3  
7-5-8
```

```
After moving 2 into the empty space
```

```
2-e-4  
1-6-3  
7-5-8
```

```
After moving 4 into the empty space
```

```
2-4-e  
1-6-3  
7-5-8
```

```
After moving 3 into the empty space
```

```
2-4-3  
1-6-e  
7-5-8
```

```
After moving 6 into the empty space
```

```
2-4-3  
1-e-6  
7-5-8
```

```
After moving 4 into the empty space
```

```
2-e-3  
1-4-6  
7-5-8
```

```
After moving 2 into the empty space
```

```
e-2-3  
1-4-6  
7-5-8
```

```
After moving 1 into the empty space
```

```
1-2-3  
e-4-6  
7-5-8
```

```
After moving 4 into the empty space
```

```
1-2-3  
4-e-6  
7-5-8
```

```
After moving 5 into the empty space
```

```
1-2-3  
4-5-6  
7-e-8
```

```
After moving 8 into the empty space. Goal achieved!
```

```
1-2-3  
4-5-6  
7-8-e
```

## CONCLUSION:

We have successfully implemented an 8-puzzle problem solver using Heuristic search technique.

## EXPERIMENT NO – 05

**AIM:** Implement Constraint Satisfaction Problem

**OBJECTIVE:** To implement the constraint Satisfaction Problem using backtracking

### DESCRIPTION:

CSPs are basically mathematical problems that are defined as a set of variables that must satisfy a number of constraints. When we arrive at the final solution, the states of the variables must obey all the constraints. This technique represents the entities involved in a given problem as a collection of a fixed number of constraints over variables. These variables need to be solved by constraint satisfaction methods. Let's use the Constraint Satisfaction framework to solve the region-colouring problem.

### ALGORITHM:

Step-1: Start

Step-2: Define a function `constraint_func(names, values)`: Step-2.1: `return values[0] != values[1]`

Step-3: Declare names

Step-4: Declare colors

Step-5: Declare constraints

Step-6: `problem = CspProblem(names, colors, constraints)`

Step-7: `output = backtrack(problem)`

Step-8: `print('\nColor mapping:\n')`

Step-9: `for k, v in output.items():`

    Step-9.1: `print(k, '==>', v)`

Step-10: End

### PROGRAM:

```
from simpleai.search import CspProblem, backtrack
```

```
def constraint_func(names, values):
```

```
    return values[0] != values[1]
```

```
if __name__ == '__main__':
```

```
    names = ('Ma', 'Ju', 'St', 'Am', 'Br',
```

```
            'Jo', 'De', 'Al', 'Mi', 'Ke')
```

```
colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in names)
constraints = [
```

```

        (('Ma', 'Ju'), constraint_func),
        (('Ma', 'St'), constraint_func),
        (('Ju', 'St'), constraint_func),
        (('Ju', 'Am'), constraint_func),
        (('Ju', 'De'), constraint_func),
        (('Ju', 'Br'), constraint_func),
        (('St', 'Am'), constraint_func),
        (('St', 'Al'), constraint_func),
        (('St', 'Mi'), constraint_func),
        (('Am', 'Mi'), constraint_func),
        (('Am', 'Jo'), constraint_func),
        (('Am', 'De'), constraint_func),
        (('Br', 'De'), constraint_func),
        (('Br', 'Ke'), constraint_func),
        (('Jo', 'Mi'), constraint_func),
        (('Jo', 'Am'), constraint_func),
        (('Jo', 'De'), constraint_func),
        (('Jo', 'Ke'), constraint_func),
        (('De', 'Ke'), constraint_func),
    ]

    problem = CspProblem(names, colors, constraints)
    output = backtrack(problem)
    print('\nColor mapping:\n')
    for k, v in output.items():
        print(k, '==>', v)

```

## OUTPUT:

```
cselab7-09@cselab709-OptiPlex-3050:~$ python constraint.py  
  
Color mapping:  
Ma ==> red  
Ju ==> green  
St ==> blue  
Am ==> red  
Br ==> red  
Jo ==> green  
De ==> blue  
Al ==> red  
Mi ==> gray  
Ke ==> gray
```

### CONCLUSION:

We have successfully implemented the Constraint Satisfaction Problem using backtracking

## **EXPERIMENT NO – 06**

**AIM:** To implement a program for game search.

**OBJECTIVE:** To implement a program for game search using minimax and alpha beta pruning.

### **DESCRIPTION:**

The Minimax algorithm is a decision-making algorithm used in game theory and artificial intelligence. It is used to determine the best move for a player in a two-player zero-sum game where the outcome depends on the choices made by both players. The algorithm works by exploring the game tree to a certain depth, assigning a score to each possible outcome, and selecting the move that minimizes the maximum loss. The algorithm alternates between maximizing and minimizing phases, and while it is effective for determining the best move, it can be computationally expensive for large game trees. Improvements such as alpha-beta pruning can be used to improve its efficiency.

Alpha-beta pruning is an optimization technique used in the Minimax algorithm to reduce the number of nodes that need to be explored in a game tree. It maintains two values, alpha and beta, to represent the best score achievable by the maximizer and the minimizer, respectively. If the algorithm finds a move that leads to a worse outcome than the current alpha or beta value, it can safely prune that part of the tree because it will not be chosen. This allows the algorithm to skip large portions of the game tree and significantly reduces the number of nodes that need to be explored, making the algorithm more efficient.

### **ALGORITHM:**

#### **Minimax**

Step-1: Start

Step-2: Construct the game tree representing all possible moves and outcomes of the game.

Step-3: Assign a score to each terminal node of the tree (i.e., the leaves) based on the outcome of the game.

Step-4: Work back up the tree from the leaves to the root, assigning scores to

each non-terminal node based on the scores of its children.

Step-5: For each maximizing node, choose the child with the highest score as the optimal move.

Step-6: For each minimizing node, choose the child with the lowest score as the optimal move.

Step-7: Continue recursively until the root node is reached, which represents the optimal move for the current player.

Step-8 : END

### **Alpha-Beta Pruning:**

Step-1: START

Step-2: Construct the game tree representing all possible moves and outcomes of the game.

Step-3: Assign a score to each terminal node of the tree (i.e., the leaves) based on the outcome of the game.

Step-4: Work back up the tree from the leaves to the root, assigning scores to each non-terminal node based on the scores of its children.

Step-5: For each maximizing node, choose the child with the highest score as the optimal move and update the alpha value accordingly.

Step-6: For each minimizing node, choose the child with the lowest score as the optimal move and update the beta value accordingly.

Step-7: If the alpha value becomes greater than or equal to the beta value at any point during the search, prune the remaining children of that node and return the current score.

Step-8: Continue recursively until the root node is reached, which represents the optimal move for the current player.

### **PROGRAM:**

```
import math
```

```
def minimax (curDepth, nodeIndex,  
            maxTurn, scores,
```



```

    targetDepth):

    if (curDepth == targetDepth):
        return scores[nodeIndex]

    if (maxTurn):

        return max(minimax(curDepth + 1, nodeIndex * 2,
                           False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                           False, scores, targetDepth))

    else:

        return min(minimax(curDepth + 1, nodeIndex * 2,
                           True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1,
                           True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]

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

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

```

### OUTPUT:

```

The optimal value is : 12
>>> |

```

### Alpha-Beta Pruning:

MAX, MIN = 1000, -1000

```

def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

```

```

best = MIN
for i in range(0, 2):

    val = minimax(depth + 1, nodeIndex * 2 + i,
                  False, values, alpha, beta)
    best = max(best, val)
    alpha = max(alpha, best)

    if beta <= alpha:
        break
return best

else:

    best = MAX

    for i in range(0, 2):

        val = minimax(depth + 1, nodeIndex * 2 + i,
                      True, values, alpha, beta)
        best = min(best, val)
        beta = min(beta, best)

        if beta <= alpha:
            break

    return best

values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

```

### OUTPUT:

```

The optimal value is : 5
>>> |

```

### CONCLUSION:

We have successfully implemented a program for game search and built a bot to implement any game using easy AI library.

## **EXPERIMENT NO – 07**

**AIM:** Implement Bayesian network

**OBJECTIVE:** Implement a Bayesian network from a given data and Infer the data from the Bayesian network

### **DESCRIPTION:**

Bayesian Networks are used to model uncertainties by using Directed Acyclic Graphs (DAG). A Directed Acyclic Graph is used to represent a Bayesian Network and like any other statistical graph, a DAG contains a set of nodes and links, where the links denote the relationship between the nodes. A DAG models the uncertainty of an event occurring based on the Conditional Probability Distribution (CPD) of each random variable. A Conditional Probability Table (CPT) is used to represent the CPD of each variable in the network

### **ALGORITHM:**

Step-1: Start

Step-2: Import necessary packages and modules like numpy, pandas, pgmpy, urllib, etc.

Step-3: Define the column names of the dataset in a list.

Step-4: Read the heart disease dataset using pandas read\_csv function and replace missing values denoted by '?' with NaN.

Step-5: Define a Bayesian network model using pgmpy's BayesianModel class and specify the directed edges between the nodes.

Step-6: Fit the model to the dataset using MaximumLikelihoodEstimator estimator.

Step-7: Create an instance of VariableElimination class from pgmpy.inference module.

Step-8: Query the model to find the probability distribution of the target variable given the evidence that age is 37, using the query method of the HeartDisease\_infer instance.

Step-9: Print the query result.

Step-10: End

### **PROGRAM:**

```
import numpy as np

from urllib.request import urlopen
```

```
import urllib
import pandas as pd
```

```

import pgmpy

from pgmpy.inference import VariableElimination
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator,
BayesianEstimator

names = ['age', 'chol', 'fbs', 'restecg', 'thalach', 'target']

heartDisease = pd.read_csv('D:\\Engineering\\SEM-
6\\Artificial Intelligence\\Lab Work\\Programs\\heart_disease_data.csv')
heartDisease = heartDisease.replace('?', np.nan)

model = BayesianModel([('age', 'fbs'), ('fbs', 'target'), ('target', 'restecg'),
('target', 'thalach'), ('target',
'chol')])

model.fit(heartDisease,
estimator=MaximumLikelihoodEstimator) from pgmpy.inference
import VariableElimination HeartDisease_infer =
VariableElimination(model)
q = HeartDisease_infer.query(variables=['target'], evidence={'age': 37})
print(q)

```

### OUTPUT:

```

+-----+-----+
| target | phi(target) |
+=====+=====+
| target(0) | 0.4496 |
+-----+-----+
| target(1) | 0.5504 |
+-----+-----+

```

### CONCLUSION:

We have successfully implemented a Bayesian network from a given data and infer the data from the Bayesian network

## **EXPERIMENT NO – 08(a)**

**AIM:** Implement a MDP to run value iteration

**OBJECTIVE:** To implement a MDP to run value iteration in any environment

### **DESCRIPTION:**

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems in situations where outcomes are partly random and partly under the control of a decision-maker. The goal of MDP is to find the optimal policy for a given environment, which is a mapping from states to actions that maximizes the expected total reward received by the agent.

Value iteration is an algorithm used to solve MDPs by iteratively computing the optimal values of each state. The algorithm starts with an initial guess for the value of each state and repeatedly updates them until convergence. At each iteration, the algorithm applies the Bellman update equation, which expresses the value of a state as the sum of the immediate reward and the discounted value of the next state. The discount factor is a parameter that determines the relative importance of immediate and future rewards.

### **ALGORITHM:**

Step-1: Start

Step-2: Initialize the parameters including reward, discount, maximum error, number of actions, actions, number of rows, number of columns, and the utility matrix U.

Step-3: Define a function named 'printEnvironment' that takes the array and policy as parameters, then iterates over the rows and columns of the array, and prints the corresponding values according to the given conditions.

Step-4: Define a function named 'getU' that takes U, row number r, column number c, and action as parameters. It then calculates the new row and column number after applying the given action and returns the value of the utility at that position. If the position is outside the matrix or corresponds to the wall, then it returns the utility value of the current position.

Step-5: Define a function named 'calculateU' that takes U, row number r, column number c, and action as parameters. It calculates the utility for the given action at the given position using the formula provided.

Step-6: Define a function named 'valueIteration' that takes U as a parameter. It iteratively updates the utility matrix by calculating the maximum utility over all

actions at each position until the maximum error is less than the specified



threshold. It prints the intermediate utilities using the 'printEnvironment' function.

Step-7: Define a function named 'getOptimalPolicy' that takes U as a parameter. It calculates the optimal policy by finding the action that gives the maximum utility at each position.

Step-8: Print the initial utility matrix using the 'printEnvironment' function.

Step-9: Call the 'valueIteration' function with the initial utility matrix U and obtain the final utility matrix.

Step-10: Call the 'getOptimalPolicy' function with the final utility matrix to get the optimal policy.

Step-11: Print the optimal policy using the 'printEnvironment' function. Step-12: End

### **PROGRAM:**

```
REWARD = -0.01
```

```
DISCOUNT = 0.99
```

```
MAX_ERROR = 10**(-3)
```

```
NUM_ACTIONS = 4
```

```
ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]
```

```
NUM_ROW = 3
```

```
NUM_COL = 4
```

```
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
```

```
def printEnvironment(arr, policy=False):
```

```
    res = ""
```

```
    for r in range(NUM_ROW):
```

```
        res += "|"
```

```
        for c in range(NUM_COL):
```

```
            if r == c == 1:
```

```
                val = "WALL"
```

```
            elif r <= 1 and c == 3:
```

```
                val = "+1" if r == 0 else "-1"
```

```
            else:
```

```
                if policy:
```

```
                    val = ["Down", "Left", "Up", "Right"][arr[r][c]]
```

```
                else:
```

```
                    val = str(arr[r][c])
```

```
    res += " " + val[:5].ljust(5) + " |"  
    res += "\n"  
print(res)
```

```

def getU(U, r, c, action):
    dr, dc = ACTIONS[action]
    newR, newC = r+dr, c+dc
    if newR < 0 or newC < 0 or newR >= NUM_ROW or newC >= NUM_COL or
    (newR == newC == 1):
        return U[r][c]
    else:
        return U[newR][newC]
def calculateU(U, r, c,
action):
    u = REWARD

    u += 0.1 * DISCOUNT * getU(U, r, c, (action-1)%4)
    u += 0.8 * DISCOUNT * getU(U, r, c, action)
    u += 0.1 * DISCOUNT * getU(U, r, c, (action+1)%4)
    return u
def valueIteration(U):
    print("During the value iteration:\n")
    while True:
        nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

        error = 0

        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                nextU[r][c] = max([calculateU(U, r, c, action) for action in
range(NUM_ACTIONS)])
                error = max(error, abs(nextU[r][c]-U[r][c]))

        U = nextU
        printEnvironment(U)
        if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
            break
    return U
def getOptimalPolicy(U):
    policy = [[-1, -1, -1, -1] for i in range(NUM_ROW)]
    for r in range(NUM_ROW):
        for c in range(NUM_COL):
            if (r <= 1 and c == 3) or (r == c == 1):
                continue

```

```
maxAction, maxU = None, -float("inf")
for action in range(NUM_ACTIONS):
    u = calculateU(U, r, c, action)
```

```

    if u > maxU:
        maxAction, maxU = action, u
    policy[r][c] = maxAction
return policy
print("The initial U is:\n")
printEnvironment(U)
U = valueIteration(U)

policy = getOptimalPolicy(U)
print("The optimal policy is:\n")
printEnvironment(policy, True)

```

## OUTPUT:

```

The initial U is:
| 0 | 0 | 0 | +1 |
| 0 | WALL | 0 | -1 |
| 0 | 0 | 0 | 0 |

During the value iteration:

| -0.01 | -0.01 | 0.782 | +1 |
| -0.01 | WALL | -0.01 | -1 |
| -0.01 | -0.01 | -0.01 | -0.01 |

| -0.01 | 0.607 | 0.858 | +1 |
| -0.01 | WALL | 0.509 | -1 |
| -0.01 | -0.01 | -0.01 | -0.01 |

| 0.467 | 0.790 | 0.917 | +1 |
| -0.02 | WALL | 0.621 | -1 |
| -0.02 | -0.02 | 0.389 | -0.02 |

| 0.659 | 0.873 | 0.934 | +1 |
| 0.354 | WALL | 0.679 | -1 |
| -0.03 | 0.292 | 0.476 | 0.196 |

| 0.781 | 0.902 | 0.941 | +1 |
| 0.582 | WALL | 0.698 | -1 |
| 0.295 | 0.425 | 0.576 | 0.287 |

| 0.840 | 0.914 | 0.944 | +1 |
| 0.724 | WALL | 0.705 | -1 |
| 0.522 | 0.530 | 0.613 | 0.375 |

| 0.869 | 0.919 | 0.945 | +1 |
| 0.798 | WALL | 0.708 | -1 |
| 0.667 | 0.580 | 0.638 | 0.414 |

```

```

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

| 0.903 | 0.930 | 0.954 | +1 |
| 0.879 | WALL | 0.789 | -1 |
| 0.853 | 0.830 | 0.805 | 0.639 |

The optimal policy is:

| Right | Right | Right | +1 |
| Up | WALL | Left | -1 |
| Up | Left | Left | Down |

```

## CONCLUSION:

We have successfully implemented a MDP to run value iteration in any environment

## **EXPERIMENT NO – 08(b)**

**AIM:** Implement a MDP to run policy iteration

**OBJECTIVE:** To implement a MDP to run policy iteration in any environment

### **DESCRIPTION:**

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making situations in which an agent interacts with an environment that is affected by random events. Policy iteration is a technique used to solve an MDP and find the optimal policy for an agent.

### **ALGORITHM:**

Step-1: Start

Step-2: Define the constants and variables used in the MDP, including the reward, discount factor, maximum error, number of actions, action space, number of rows and columns, and initial utility and policy matrices.

Step-3: Define the printEnvironment function to print the current state of the environment, including the utility values or actions at each grid cell.

Step-4: Define the getU function to get the utility value at the given grid cell and action.

Step-5: Define the calculateU function to calculate the utility value at the given grid cell and action.

Step-6: Define the policyEvaluation function to evaluate the current policy and calculate the utility values for each grid cell.

Step-7: Define the policyIteration function to perform the policy iteration algorithm, which alternates between policy evaluation and policy improvement until the policy converges to an optimal policy.

Step-8: Print the initial random policy.

Step-9: Call the policyIteration function with the initial policy and utility matrices as inputs.

Step-10: Print the optimal policy.

Step-11: End

### **PROGRAM:**

```
import random
REWARD = -0.01
DISCOUNT = 0.99
```

```
MAX_ERROR = 10**(-3)
NUM_ACTIONS = 4
```

```

ACTIONS = [(1, 0), (0, -1), (-1, 0), (0, 1)]
NUM_ROW = 3
NUM_COL = 4
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
policy = [[random.randint(0, 3) for j in range(NUM_COL)] for i in range(NUM_ROW)]
def printEnvironment(arr, policy=False):
    res = ""
    for r in range(NUM_ROW):
        res += "|"
        for c in range(NUM_COL):
            if r == c == 1:
                val = "WALL"
            elif r <= 1 and c == 3:
                val = "+1" if r == 0 else "-1"
            else:
                val = ["Down", "Left", "Up", "Right"][arr[r][c]]
            res += " " + val[:5].ljust(5) + " |"
        res += "\n"
    print(res)
def getU(U, r, c, action):
    dr, dc = ACTIONS[action]
    newR, newC = r+dr, c+dc
    if newR < 0 or newC < 0 or newR >= NUM_ROW or newC >= NUM_COL or (newR == newC == 1):
        return U[r][c]
    else:
        return U[newR][newC]
def calculateU(U, r, c, action):
    u = REWARD
    u += 0.1 * DISCOUNT * getU(U, r, c, (action-1)%4)
    u += 0.8 * DISCOUNT * getU(U, r, c, action)
    u += 0.1 * DISCOUNT * getU(U, r, c, (action+1)%4)
    return u
def policyEvaluation(policy, U):
    while True:
        nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

```



```
error = 0  
for r in range(NUM_ROW):  
    for c in range(NUM_COL):
```

```

        if (r <= 1 and c == 3) or (r == c == 1):
            continue
        nextU[r][c] = calculateU(U, r, c, policy[r][c])
        error = max(error, abs(nextU[r][c]-U[r][c]))

    U = nextU

    if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
        break
    return U
def policyIteration(policy, U):
    print("During the policy iteration:\n")
    while True:
        U = policyEvaluation(policy, U)
        unchanged = True
        for r in range(NUM_ROW):
            for c in range(NUM_COL):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                maxAction, maxU = None, -float("inf")
                for action in range(NUM_ACTIONS):
                    u = calculateU(U, r, c, action)
                    if u > maxU:
                        maxAction, maxU = action, u

                if maxU > calculateU(U, r, c, policy[r][c]):
                    policy[r][c] = maxAction
                    unchanged = False
            if unchanged:
                break
        printEnvironment(policy)
    return policy

print("The initial random policy is:\n")
printEnvironment(policy)
policy = policyIteration(policy, U)
print("The optimal policy is:\n")
printEnvironment(policy)

```

## OUTPUT:

```
| Up    | Up    | Left  | +1    |
| Up    | WALL  | Down  | -1    |
| Right | Down  | Right | Left  |

During the policy iteration:

| Left  | Left  | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Up    | Left  | Left  | Down  |

| Left  | Right | Right | +1    |
| Up    | WALL  | Up    | -1    |
| Up    | Right | Up    | Down  |

| Right | Right | Right | +1    |
| Down  | WALL  | Left  | -1    |
| Right | Right | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Right | Right | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Right | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Up    | Down  |

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |

The optimal policy is:

| Right | Right | Right | +1    |
| Up    | WALL  | Left  | -1    |
| Up    | Left  | Left  | Down  |
```

## CONCLUSION:

We have successfully implemented a MDP to run policy iteration in any environment.

## Viva Questions

### A\* Algorithm

1. What is the A\* algorithm, and how does it work?
2. What is the difference between breadth-first search and A\* algorithm?
3. Can you explain the concept of heuristics in A\* algorithm?
4. What is the role of the cost function in A\* algorithm?
5. What are the advantages and disadvantages of using A\* algorithm?
6. How does the A\* algorithm handle cases where the optimal solution is not found?
7. How can you optimize the performance of the A\* algorithm?
8. Can you give an example of a problem that can be solved using A\* algorithm?
9. How does the A\* algorithm handle complex or non-linear problem spaces?
10. What are some variations of the A\* algorithm, and when would you use them?

### Water Jug Problem

1. What is the Water Jug Problem?
2. Can you describe the steps involved in solving the Water Jug Problem?
3. How can you determine whether the Water Jug Problem is solvable or not?
4. Is it possible to solve the Water Jug Problem with just two jugs of different sizes? Why or why not?
5. Can you explain the concept of a "state" in the context of the Water Jug Problem?
6. What is the difference between the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in solving the Water Jug Problem?
7. Can you describe a heuristic approach to solving the Water Jug Problem?
8. How can you modify the Water Jug Problem to solve a more complex problem, such as the "Missionaries and Cannibals" problem?
9. Are there any real-world applications of the Water Jug Problem or similar problems in computer science?
10. Can you explain the limitations of the brute-force approach to solving the Water Jug Problem, and how other algorithms can overcome these limitations?

### Policy Iteration and Value Iteration

1. What is an MDP, and what are its components?
2. How do you define the state transition probabilities and rewards in an MDP?
3. What is the difference between a policy and a value function in reinforcement learning?
4. What is the goal of policy iteration, and how does it work?
5. What are the steps involved in policy iteration, and how do they relate to the Bellman equations?
6. How does policy iteration guarantee convergence to the optimal policy in a finite MDP?
7. How does the choice of initial policy affect the convergence rate of policy iteration?
8. What are some variations of policy iteration, and how do they differ from the standard algorithm?
9. How do you evaluate a policy in policy iteration, and what is the role of the value function in this process?
10. How do you select the best policy after policy iteration converges, and what is the difference between the optimal policy and the optimal value function?

### Bayesian Network

1. What is a Bayesian network and how does it represent probabilistic relationships between variables?
2. What are the main advantages of using Bayesian networks over other modeling techniques?

3. How do you learn the structure and parameters of a Bayesian network from data?
4. What is the role of prior probabilities in Bayesian network inference and how do they impact the results?
5. What are the different methods for performing inference on Bayesian networks, and when might each method be appropriate?
6. How can you use Bayesian networks to make predictions or classify new instances?
7. What are some common applications of Bayesian networks in real-world settings?
8. What are some potential limitations or challenges of using Bayesian networks in practice, and how can they be addressed?
9. How can you evaluate the performance of a Bayesian network model, and what metrics might you use?

#### **game search**

1. What is game search and why is it important in game development?
2. What is the difference between a game tree and a search tree?
3. Can you explain the minimax algorithm and how it works in game search?
4. What is alpha-beta pruning and how does it improve the efficiency of game search?
5. How do heuristic evaluation functions help guide game search?

#### **Constraint Satisfaction Problem:**

1. What is Constraint Satisfaction Problem (CSP)?
2. What is the main idea behind the backtracking algorithm for solving CSPs?
3. What is the difference between a variable and a value in CSPs?

#### **8-puzzle solver using Heuristic search technique**

1. What is the 8-puzzle problem and how does it relate to heuristic search techniques?
2. What is a heuristic function and how is it used in the 8-puzzle problem?
3. What is the Manhattan distance heuristic and how is it calculated in the 8-puzzle problem?
4. Can you explain the A\* algorithm and how it is used to solve the 8-puzzle problem?
5. What is the difference between the best-first search and A\* algorithms, and which one is more suitable for the 8-puzzle problem?
6. Can you discuss some of the advantages and disadvantages of using heuristic search techniques to solve the 8-puzzle problem?
7. How does the performance of the heuristic search technique depend on the choice of heuristic function?
8. How can you improve the efficiency of the heuristic search algorithm in solving the 8-puzzle problem?