**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

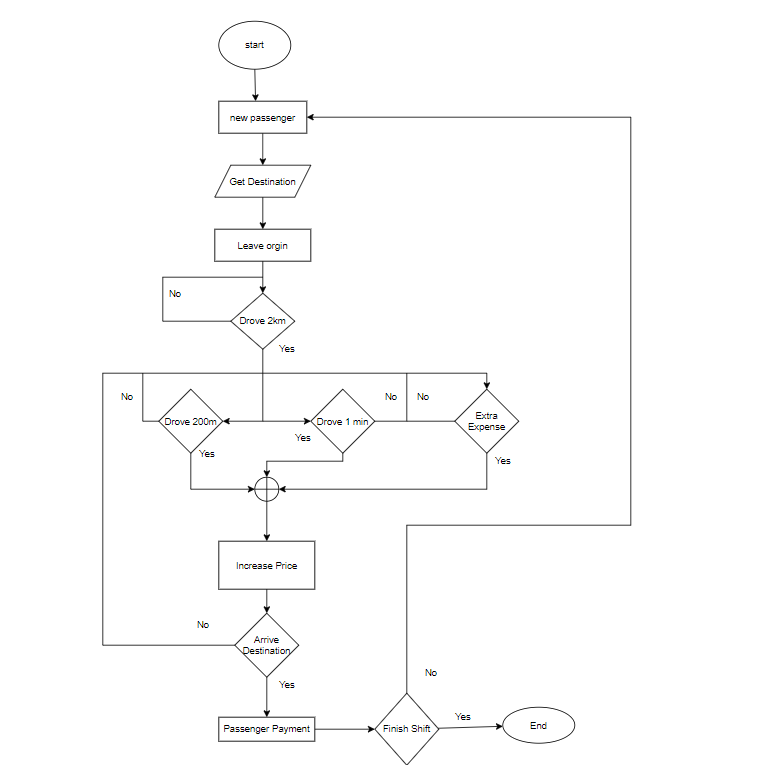
- Performance: Healthy patient, minimize costs, lawsuits

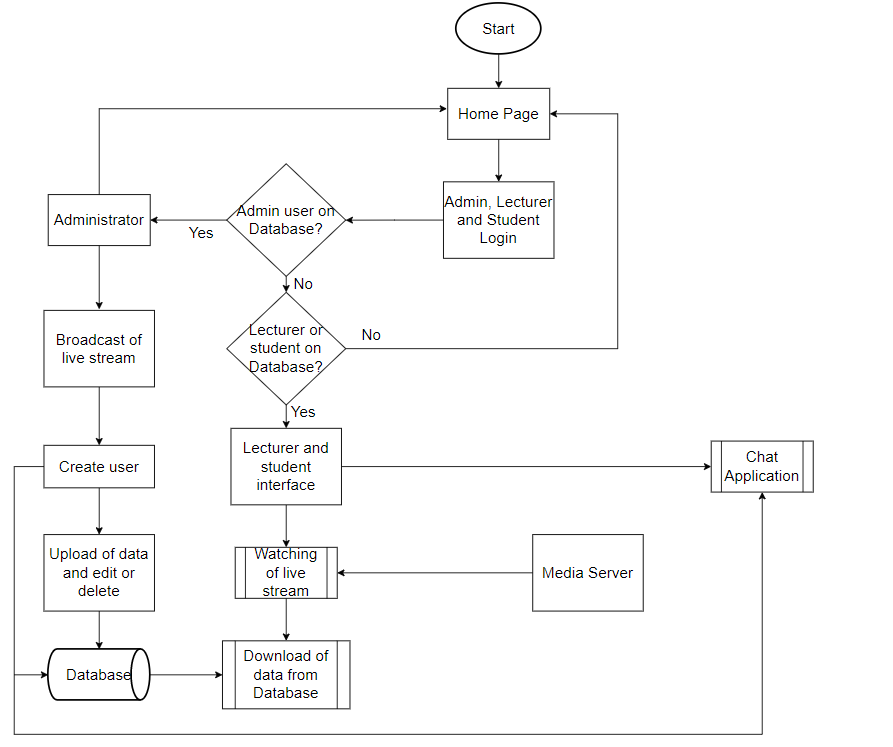
- Environment: Patient, hospital, staff

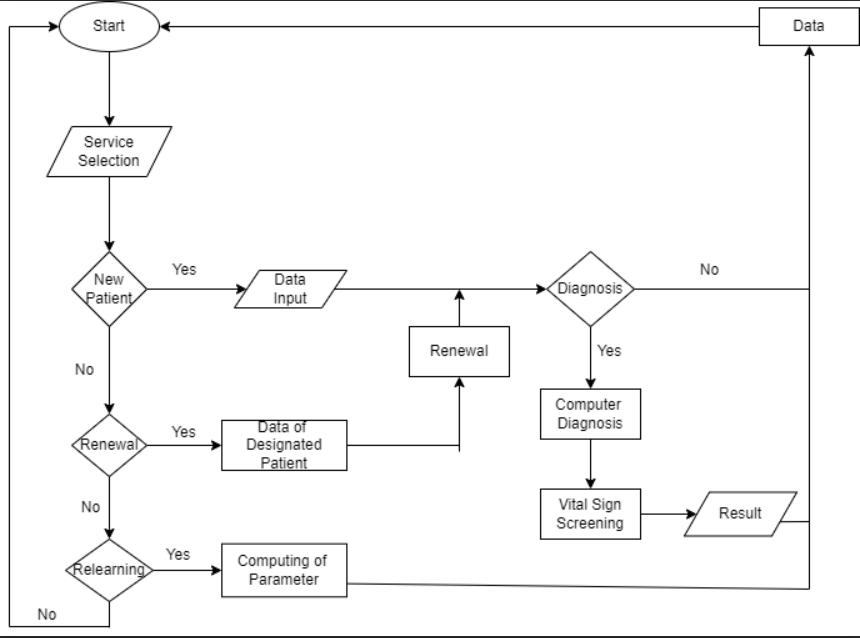
- Actuators: Screen display (questions, tests, diagnosis, treatments, referrals)

- 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 – 2**

**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: "))

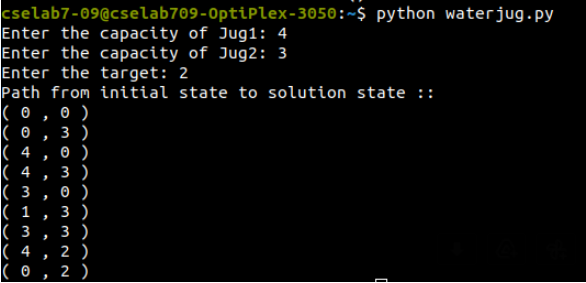
target = int(input("Enter the target: "))

print("Path from initial state "

"to solution state ::")

Solution(Jug1, Jug2, target)

**Output:**



**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:**

**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))))

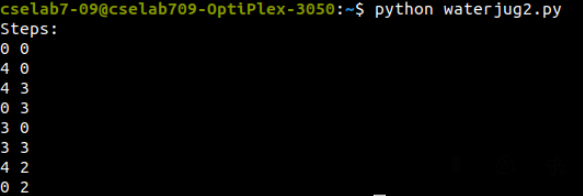
else:

return False

print("Steps: ")

waterJugSolver(0, 0)

**Output:**



**CONCLUSION:**

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

**EXPERIMENT NO – 03**

**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>. 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:

return i, j

GOAL = '''1-2-3

4-5-6

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:

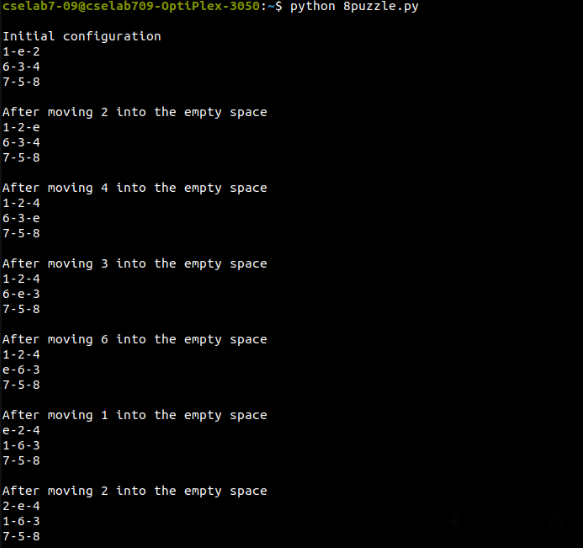
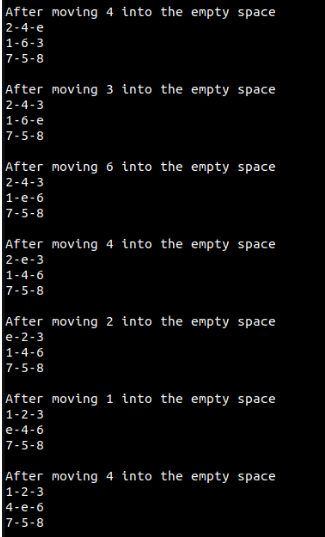
print('After moving', action, 'into the empty space. Goal achieved!')

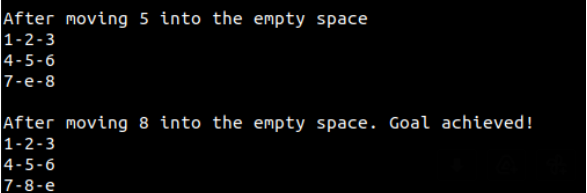
else:

print('After moving', action, 'into the empty space')

print(state)

**OUTPUT:**





**CONCLUSION:**

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

**EXPERIMENT NO – 04**

**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)

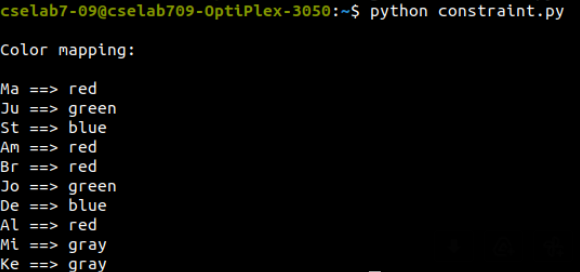
output = backtrack(problem)

print('\nColor mapping:\n')

for k, v in output.items():

print(k, '==>', v)

**OUTPUT:**



**CONCLUSION:**

We have successfully implemented the Constraint Satisfaction Problem using backtracking

**EXPERIMENT NO – 05**

**AIM:** Build a bot to implement tic-tac-toe game

**OBJECTIVE:** To implement a program for game search and build a bot to implement any game using easy AI library (ex. tic-tac- toe, game of bones).

**DESCRIPTION:**

Search algorithms are used in games to figure out a strategy. The algorithms search through the possibilities and pick the best move. There are various parameters to think about speed, accuracy, complexity, and so on. These algorithms consider all possible actions available at this time and then evaluate their future moves based on these options. The goal of these algorithms is to find the optimal set of moves that will help them arrive at the final condition. Every game has a different set of winning conditions. These algorithms use those conditions to find the set of moves.

For example, if you are playing

Tic–Tac–Toe (Noughts and Crosses), you can construct this tree to represent all possible moves. We start from the root of the tree, which is the starting point of the game. This node will have several children that represent various possible moves. Those children, in turn, will have more children that represent game states after more moves by the opponent. The terminal nodes of the tree represent the final results of the game after making various moves. The game would either end in a draw or one of the players would win it. The search algorithms search through this tree to make decisions at each step of the game.

**ALGORITHM:**

Step-1: Start

Step-2: Import the necessary classes from the easyAI library - TwoPlayerGame and Human\_Player.

Step-3: Define a TicTacToe class that inherits from TwoPlayerGame.

Step-4: In the **init** method of the TicTacToe class:

Step-4.1: Define a list of players that is passed as an argument to the constructor.

Step-4.2: Initialize the game board as a list of zeros, representing empty cells.

Step-4.3: Set the current player to 1.

Step-5: Define a possible\_moves method in the TicTacToe class that returns a list of the available moves on the game board.

Step-5.1: The method scans the game board and adds any empty cell to the list of possible moves.

Step-5.2: The available moves are represented as a list of integers between 1 and 9, where each integer represents a cell on the game board.

Step-6: Define a make\_move method in the TicTacToe class that updates the game board when a player makes a move.

Step-6.1: The method takes an integer move as an argument, which is the cell where the player wishes to place their symbol (either X or O).

Step-6.2: The method updates the game board by setting the value of the cell to the current player's index (either 1 or 2).

Step-7: Define an unmake\_move method in the TicTacToe class that undoes the most recent move made by a player.

Step-7.1: The method takes an integer move as an argument, which is the cell that was previously updated by the make\_move method.

Step-7.2: The method updates the game board by setting the value of the cell back to zero.

Step-8: Define a lose method in the TicTacToe class that checks if the current player has lost the game.

Step-8.1: The method checks if any of the eight possible winning combinations have been achieved by the opponent.

Step-8.2: Each winning combination is represented as a list of three integers, which correspond to the cells that must be filled by the opponent to win the game.

Step-9: Define an is\_over method in the TicTacToe class that checks if the game is over.

Step-9.1: The game is over if there are no more possible moves or if the current player has lost the game.

Step-10: Define a show method in the TicTacToe class that displays the current state of the game board in the console.

Step-10.1: The method iterates over each row and column of the game board and prints the corresponding symbol for each cell (either '.', 'O', or 'X').

Step-11: Define a scoring method in the TicTacToe class that assigns a score to the current game state.

Step-11.1: The method returns a score of -100 if the current player has lost the game, and 0 otherwise.

Step-12: If the current module is being run as the main program, create an instance of the TicTacToe class and pass it a list of two players:

Step-12.1: A human player, represented by an instance of the Human\_Player class.

Step-12.2: An AI player, represented by an instance of the AI\_Player class, which takes as an argument an instance of the Negamax class with a depth of 6.

Step-13: Call the play method on the TicTacToe instance to start the game. The game will continue until it is over, at which point the final game board and winner will be displayed.

Step-14: End

**PROGRAM:**

from easyAI import TwoPlayerGame

from easyAI.Player import Human\_Player

class TicTacToe(TwoPlayerGame):

def \_\_init\_\_(self, players):

self.players = players

self.board = [0 for i in range(9)]

self.current\_player = 1

def possible\_moves(self):

return [i + 1 for i, e in enumerate(self.board) if e == 0]

def make\_move(self, move):

self.board[int(move) - 1] = self.current\_player

def unmake\_move(self, move):

self.board[int(move) - 1] = 0

def lose(self):

return any(

[

all([(self.board[c - 1] == self.opponent\_index) for c in line])

for line in [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

[1, 4, 7],

[2, 5, 8],

[3, 6, 9],

[1, 5, 9],

[3, 5, 7],

]

]

)

def is\_over(self):

return (self.possible\_moves() == []) or self.lose()

def show(self):

print(

"\n"

+ "\n".join(

[

" ".join([[".", "O", "X"][self.board[3 \* j + i]] for i in range(3)])

for j in range(3)

]

)

)

def scoring(self):

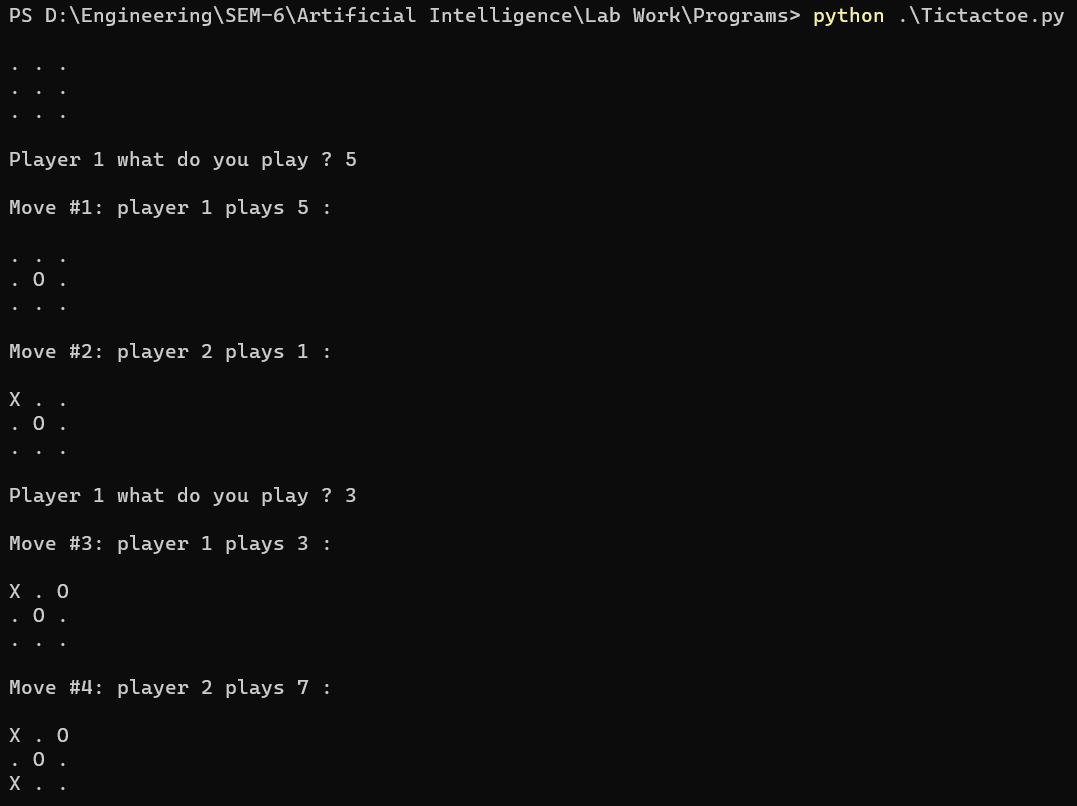
return -100 if self.lose() else 0

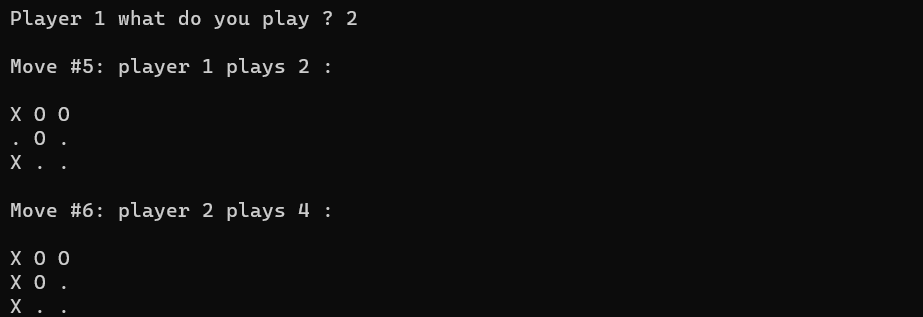
if \_\_name\_\_ == "\_\_main\_\_":

from easyAI import AI\_Player, Negamax

ai\_algo = Negamax(6)

TicTacToe([Human\_Player(), AI\_Player(ai\_algo)]).play()

**OUTPUT:**

**CONCLUSION:**

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

**EXPERIMENT NO – 06**

**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 (CDP) 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)

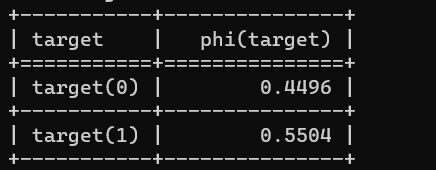
from pgmpy.inference import VariableElimination

HeartDisease\_infer = VariableElimination(model)

q = HeartDisease\_infer.query(variables=['target'], evidence={'age': 37})

print(q)

**OUTPUT:**



**CONCLUSION:**

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

**EXPERIMENT NO – 07(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)

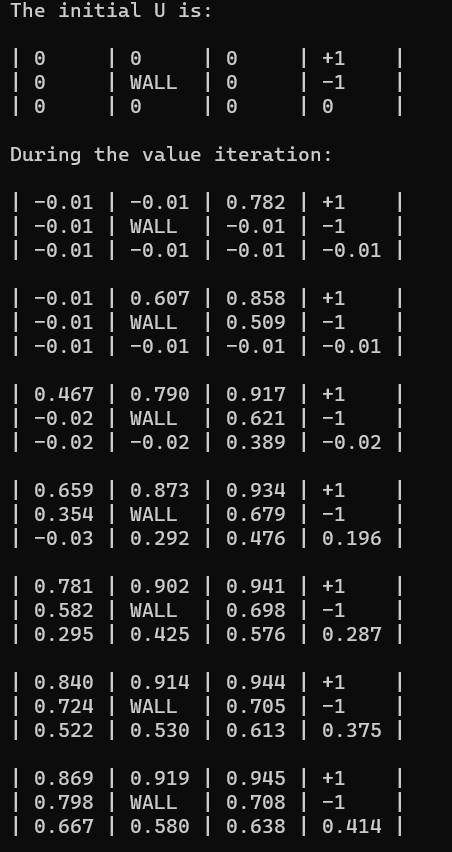
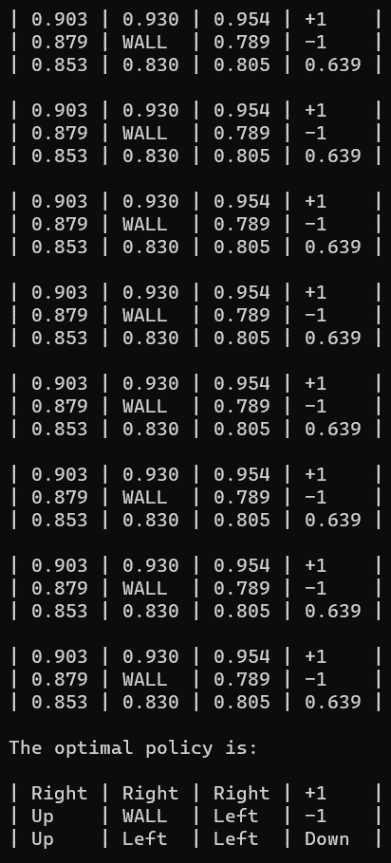
U = valueIteration(U)

policy = getOptimalPolicy(U)

print("The optimal policy is:\n")

printEnvironment(policy, True)

**OUTPUT:**



**CONCLUSION:**

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

**EXPERIMENT NO – 07(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")

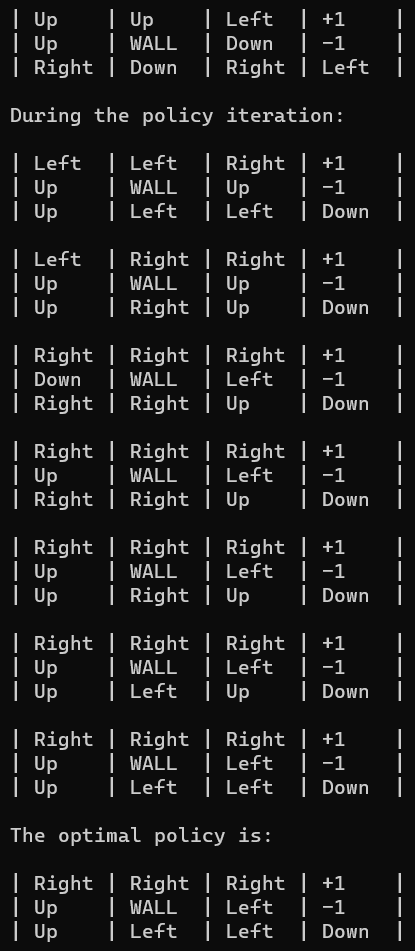
printEnvironment(policy)

policy = policyIteration(policy, U)

print("The optimal policy is:\n")

printEnvironment(policy)

**OUTPUT:**



**CONCLUSION:**

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