\_\_Artificial Intelligence Algorithms\_\_

**1. AIM: Implementation of DFS for water jug problem:**

Description:

**Water Jug Problem** is one of the most important problems to solve in Java. The water jug problem is a problem where we have two jugs, "i" liter jug and "j" liter jug (0 < i < j). Both jugs will initially be empty, and they don't have marking to measure small quantities. Now, we need to measure d liters of water by using these two jugs where d < j.

We use the following three operations to measure small quantities by using the two jars:

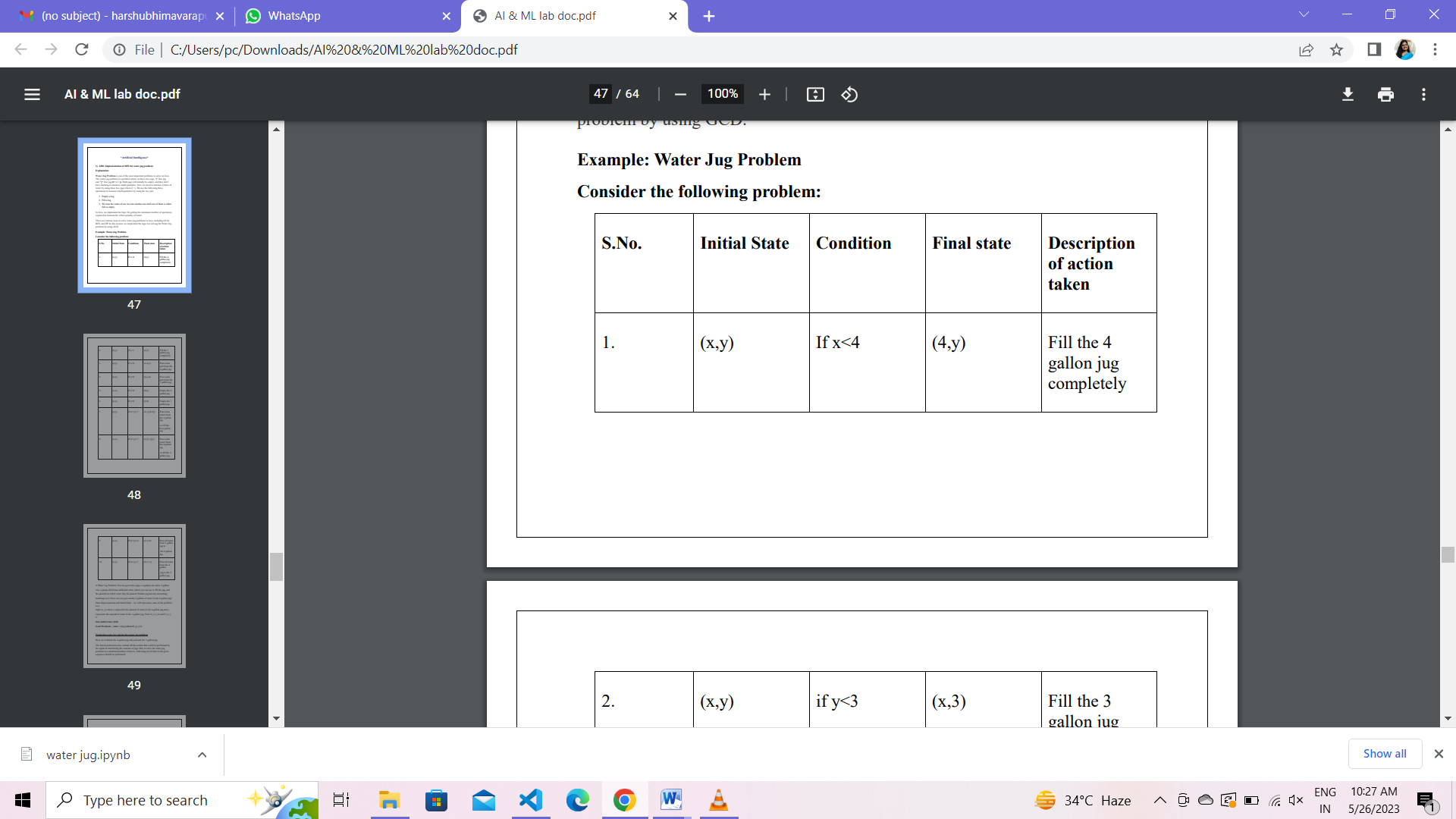
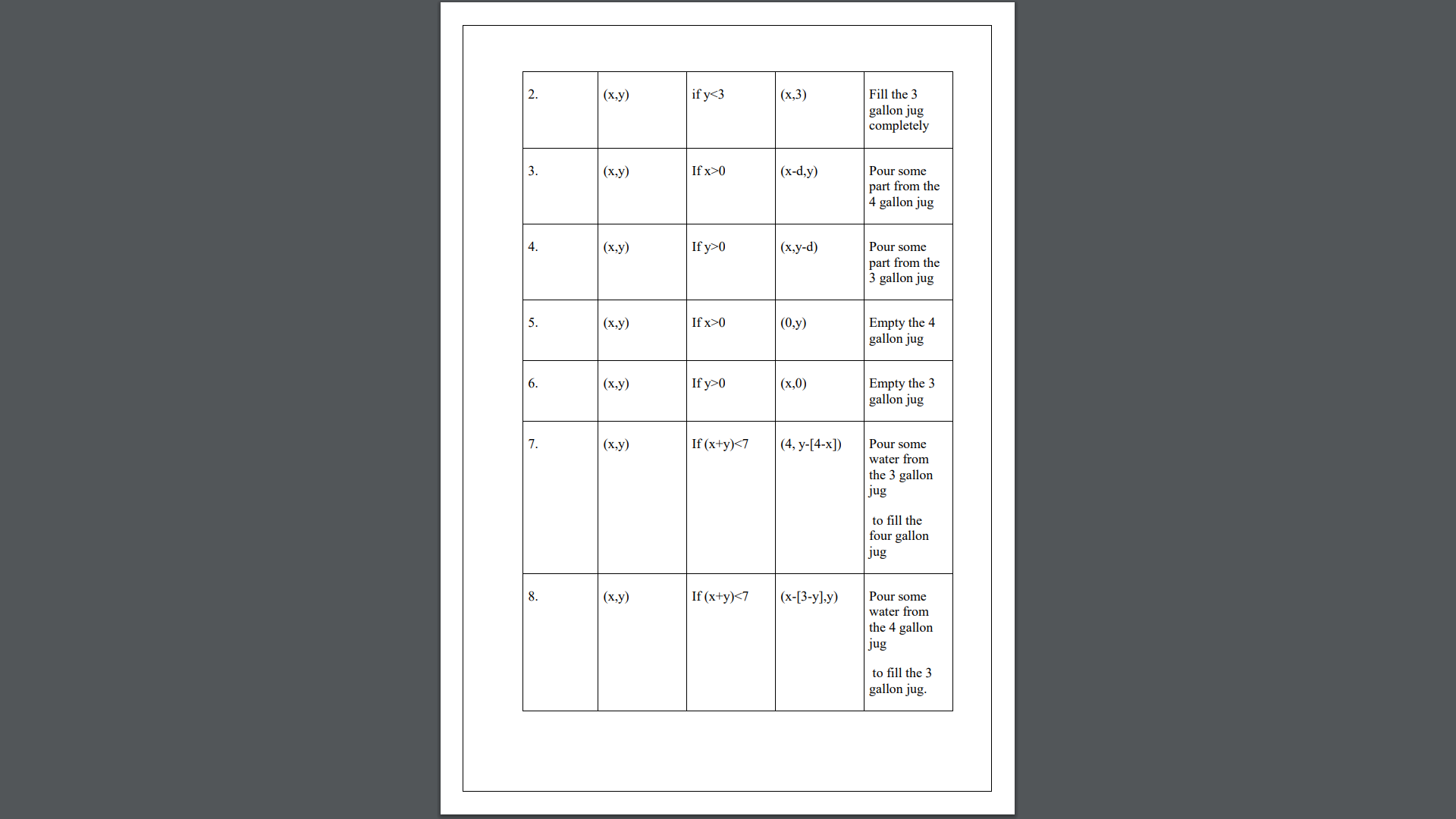
1. Empty a Jug.

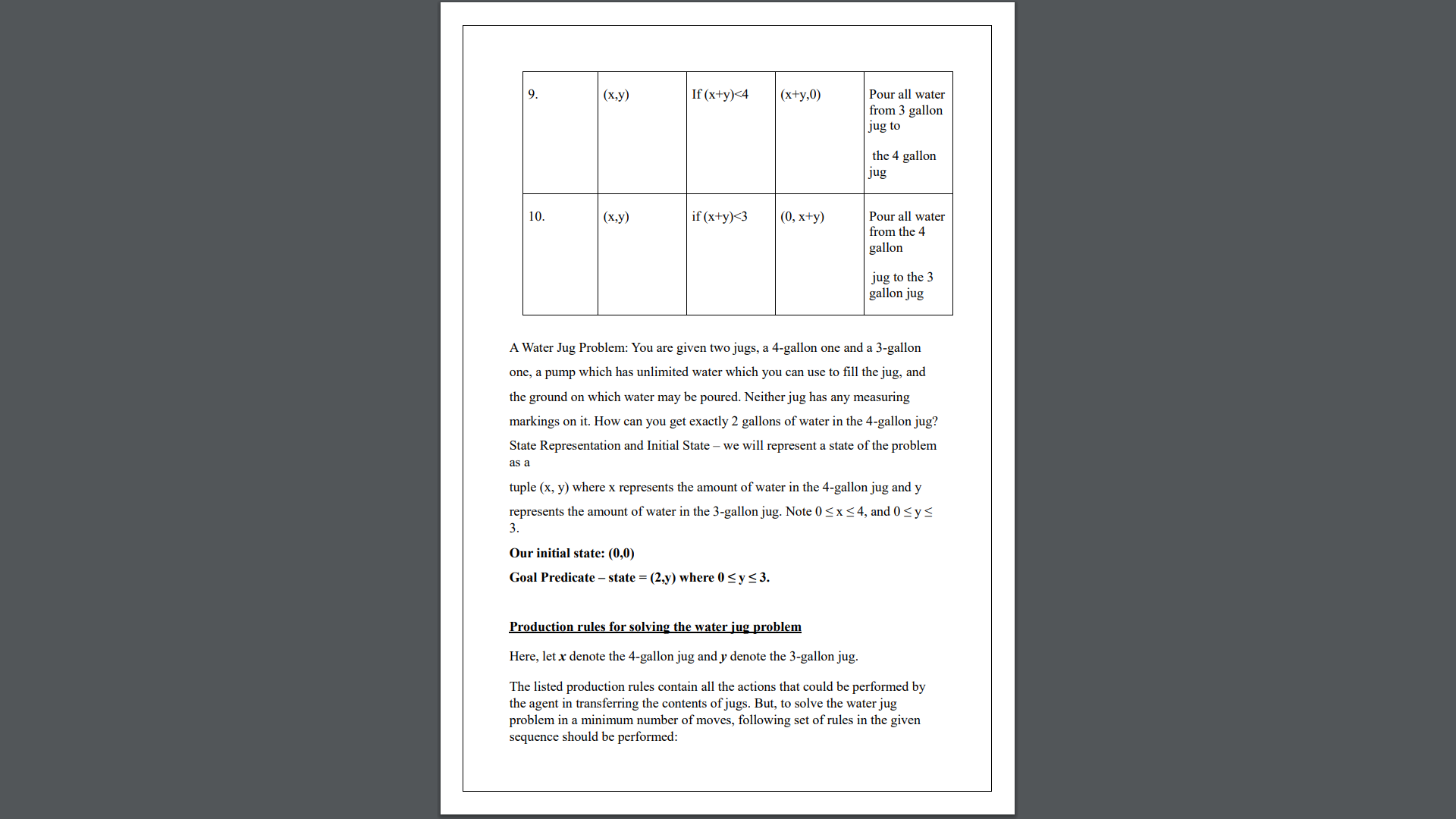
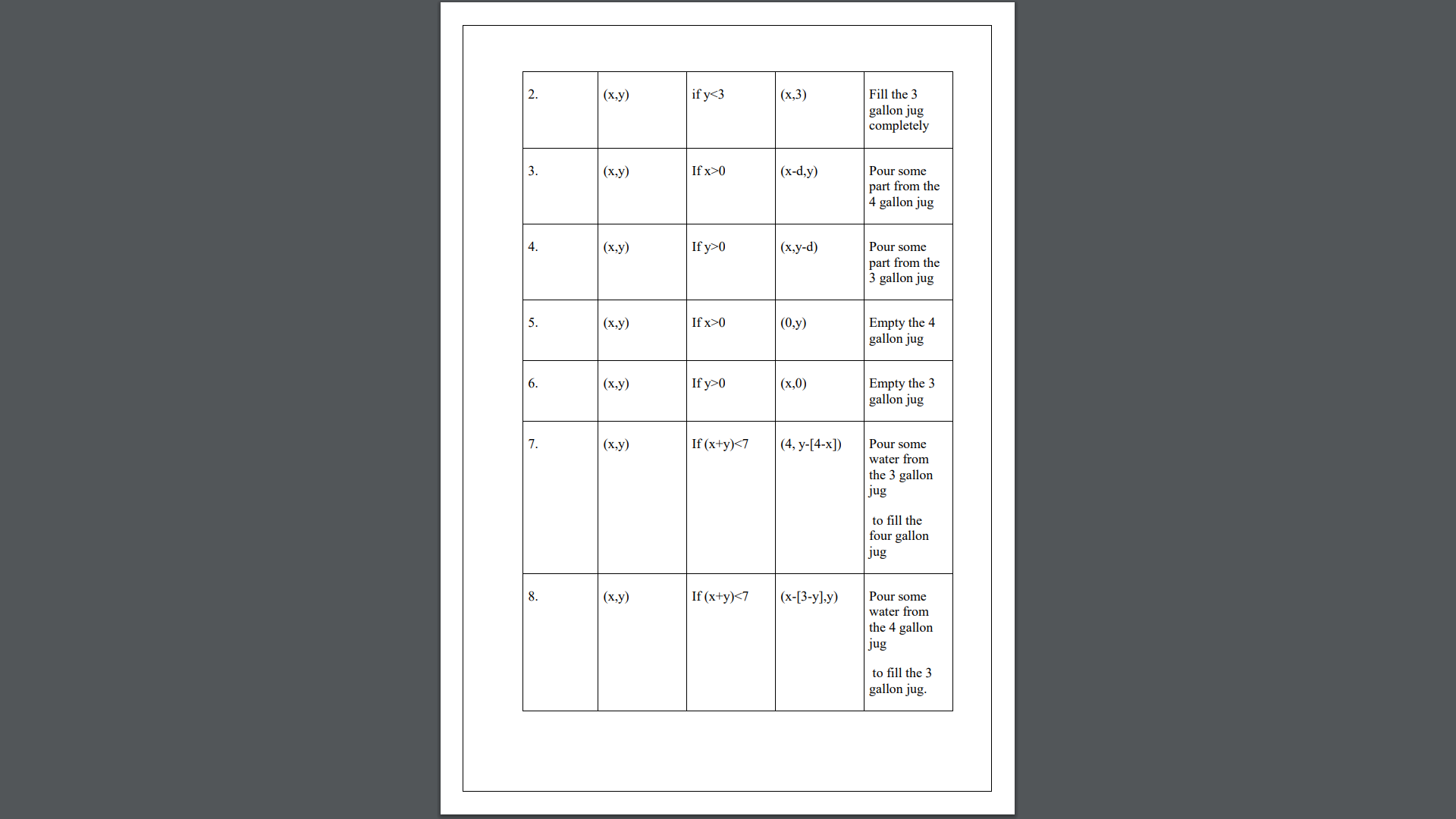
2. Fill a Jug

3. We pour the water of one jug into another one until one of them is either full or empty.

In Java, we implement the logic for getting the minimum number of operations required to measure the d liter quantity of water. There are various ways to solve water jug problems in Java, including GCD, BFS, and DP. In this section, we implement the logic for solving the Water Jug problem by using GCD

**Example:** Consider the following problem-



A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it.

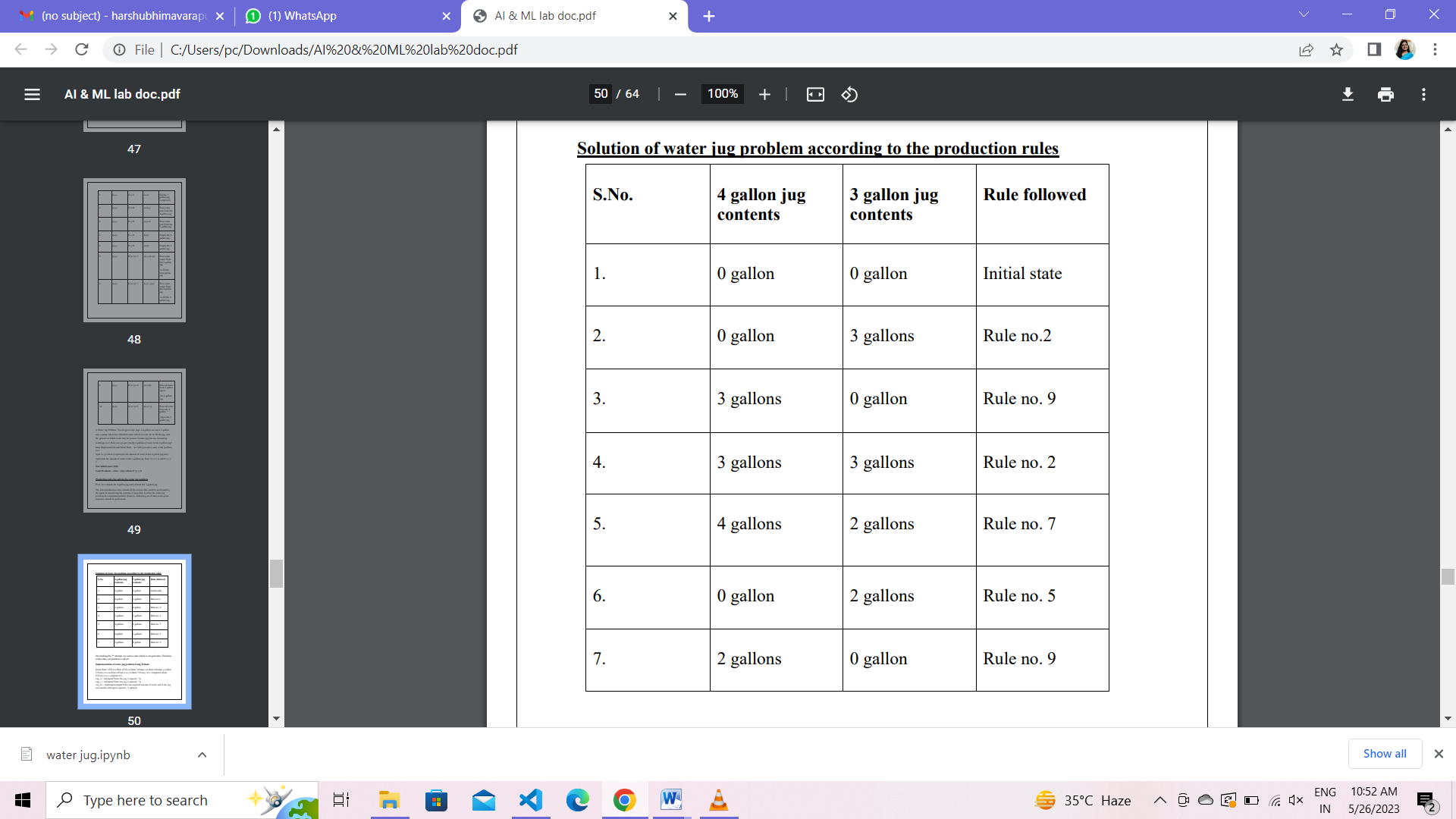
How can you get exactly 2 gallons of water in the 4-gallon jug? State Representation and Initial State – we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note 0 ≤ x ≤ 4, and 0 ≤ y ≤ 3.

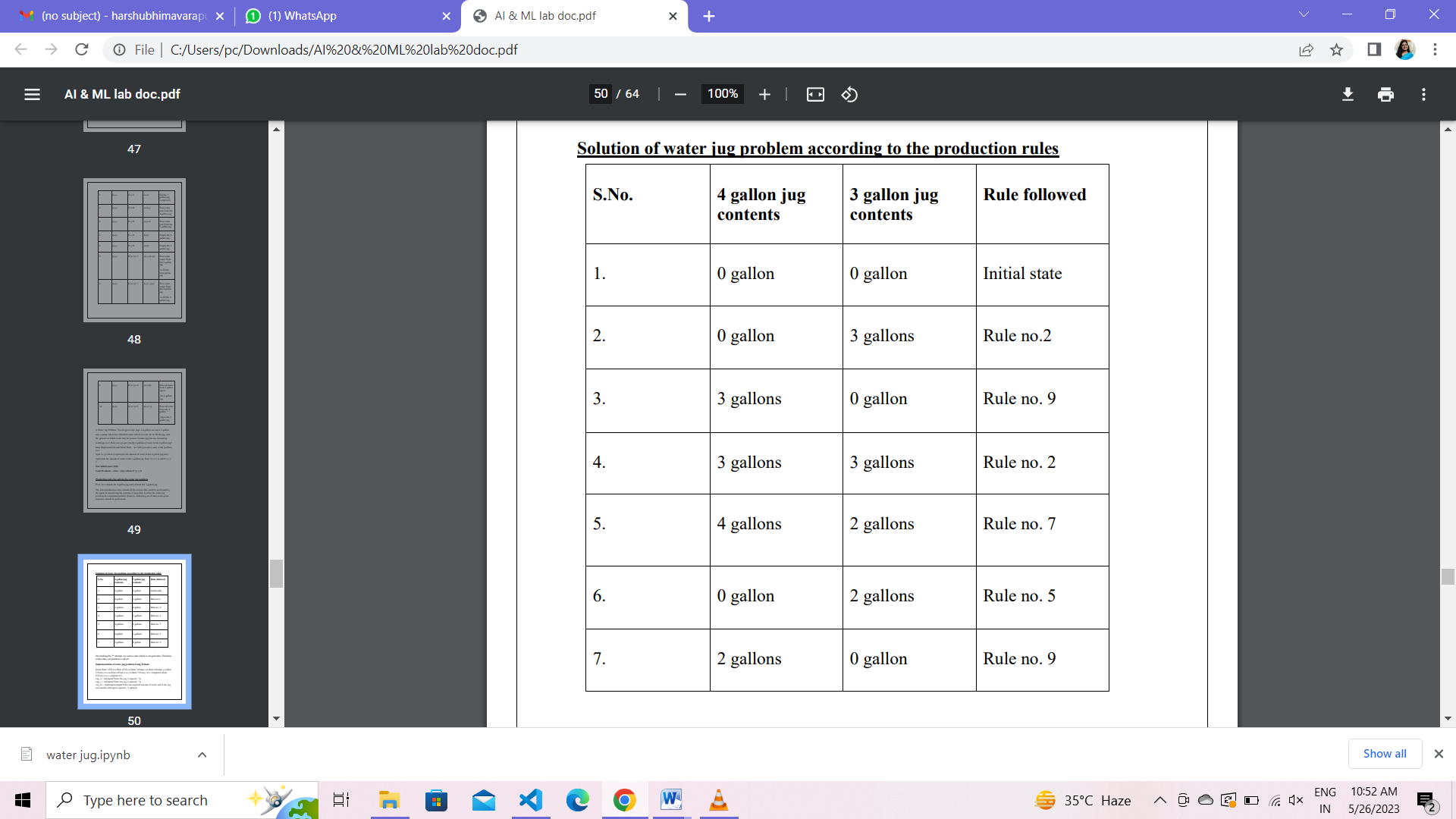
**Our initial state: (0,0)**

**Goal Predicate – state = (2,y) where 0 ≤ y ≤ 3.**

**Production rules for solving the water jug problem**

Here, let x denote the 4-gallon jug and y denote the 3-gallon jug. The listed production rules contain all the actions that could be performed by the agent in transferring the contents of jugs. But, to solve the water jug problem in a minimum number of moves, following set of rules in the given sequence should be performed:





On reaching the 7th attempt, we reach a state which is our goal state. Therefore,

at this state, our problem is solved.

**Implementation of water jug problem Using Python:**

x=int(input("Enter x-gallon jug: "))

y=int(input("Enter y-gallon jug: "))

while (True):

rule=int(input("Enter the rule should apply: "))

if rule==1:

if x<4:

x=4

if rule==2:

if y<3:

y=3

if rule==3:

if x>0:

x=0

if rule==4:

if y>0:

y=0

if rule==5:

if x+y>=4 and y>0:

x,y=4,y-(4-x)

if rule==6:

if x+y>=3 and x>0:

x,y=x-(3-y),3

if rule==7:

if x+y<=4 and y>=0:

x,y=x+y,0

if rule==8:

if x+y<=3 and x>=0:

x,y=0,x+y

print("x: ",x)

print("y: ",y)

if(x==2):

print("Goal state reached")

break

**Output:**

Enter x-gallon jug: 0

Enter y-gallon jug: 0

Enter the rule should apply: 1

x: 4

y: 0

Enter the rule should apply: 6

x: 1

y: 3

Enter the rule should apply: 4

x: 1

y: 0

Enter the rule should apply: 8

x: 0

y: 1

Enter the rule should apply: 1

x: 4

y: 1

Enter the rule should apply: 6

x: 2

y: 3

Goal state reached

**Implementation of water jug problem Using Java:**

import java.util.Scanner;

public class Main { public static void main(String[] args)

{

System.out.println("WATER JUG PROBLEM");

Scanner res=new Scanner(System.in);

System.out.println("ENTER CAPACITY OF JUG-1 :");

int x=res.nextInt();

System.out.println("ENTER CAPACITY OF JUG-2 :");

int y=res.nextInt();

System.out.println("ENTER THE GOAL STATE :");

int a=res.nextInt();

do

{

System.out.println("ENTER rule num :");

int rule=res.nextInt();

if (rule==1)

{

if (x0)

x=0;

} else if (rule==4)

{ if (y>0)

y=0;

} else if (rule==5)

{

if (x+y>=4 && y>0)

y=y-(4-x);

x=4;

} else if (rule==6)

{

if (x+y>=3 && x>0)

x=x-(3-y);

y=3;

} else if (rule==7)

{

if (x+y<=4 && y>=0)

x=x+y;

y=0;

} else if (rule==8)

{

if (x+y<=4 && y>=0)

y=x+y;

x=0;

}

//if (x==a || y==a)

//System.out.println("goal reached");

//break;

System.out.println("x= "+x);

System.out.println("y= "+y);

} while(x!=a && y!=a);

System.out.println("GOAL REACHED");

}

}

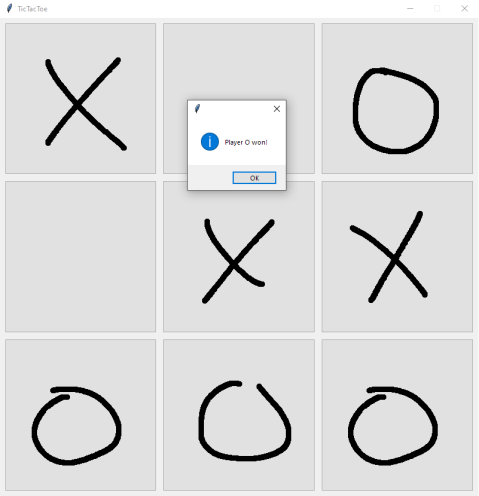
**2.Aim: Implement and demonstrate the Tic-Tac-Toe problem in python code.**

**Description:**

Tic-Tac-Toe is a classic paper-and-pencil game played on a 3x3 grid. The game is typically played by two players, who take turns marking empty cells on the grid with their respective symbols, usually 'X' and 'O'. The goal of the game is to form a horizontal, vertical, or diagonal line of three of their own symbols.

The game starts with an empty grid. Players take turns, beginning with one player (often 'X'), and place their symbol in an empty cell of the grid. The next player then takes their turn, placing their symbol in another empty cell. This process continues until one player wins or the game ends in a tie.

A player wins the game if they successfully place three of their symbols in a row, either horizontally, vertically, or diagonally. When a player wins, the game is over, and a message is typically displayed to indicate the winner. If all the cells on the grid are filled without any player achieving a winning combination, the game ends in a tie.



In the Python code example provided earlier, the Tic-Tac-Toe game is implemented using the Tkinter library for the graphical user interface. The game state is stored in a list, and the current player is tracked using a variable. Players can click on the cells of the grid to make their moves, and the game checks for wins or ties after each move. The game provides a message box to display the outcome of the game and allows players to reset and play again.

Overall, Tic-Tac-Toe is a simple yet engaging game that challenges players to strategize and think ahead to block their opponent's moves while trying to create their winning combinations.

**Implementation of Tic-tac-Toe using python code:**

**def** print\_board(board):

print('-------------')

**for** i **in** range(3):

row **=** '| '

**for** j **in** range(3):

row **+=** board[i][j] **+** ' | '

print(row)

print('-------------')

**def** check\_win(board, player):

**for** i **in** range(3):

**if** board[i][0] **==** player **and** board[i][1] **==** player **and** board[i][2] **==** player:

**return** **True**

**if** board[0][i] **==** player **and** board[1][i] **==** player **and** board[2][i] **==** player:

**return** **True**

**if** board[0][0] **==** player **and** board[1][1] **==** player **and** board[2][2] **==** player:

**return** **True**

**if** board[0][2] **==** player **and** board[1][1] **==** player **and** board[2][0] **==** player:

**return** **True**

**return** **False**

**def** tic\_tac\_toe():

board **=** [[' ' **for** \_ **in** range(3)] **for** \_ **in** range(3)]

players **=** ['X', 'O']

current\_player **=** players[0]

print\_board(board)

**while** **True**:

print(f"It's {current\_player}'s turn.")

row **=** int(input('Enter row (0-2): '))

col **=** int(input('Enter column (0-2): '))

**if** board[row][col] **!=** ' ':

print('That cell is already taken. Try again.')

**continue**

board[row][col] **=** current\_player

print\_board(board)

**if** check\_win(board, current\_player):

print(f'{current\_player} wins!')

**return**

**if all([cell != ' ' for row in board for cell in row]):**

**print("It's a tie!")**

**return**

**current\_player = players[(players.index(current\_player) + 1) % 2]**

**tic\_tac\_toe()**

**Output:**

-------------

| | | |

-------------

| | | |

-------------

| | | |

-------------

It's X's turn.

Enter row (0-2): 1

Enter column (0-2): 2

-------------

| | | |

-------------

| | | X |

-------------

| | | |

-------------

It's O's turn.

Enter row (0-2): 1

Enter column (0-2): 1

-------------

| | | |

-------------

| | O | X |

-------------

| | | |

-------------

It's X's turn.

Enter row (0-2): 0

Enter column (0-2): 2

-------------

| | | X |

-------------

| | O | X |

-------------

| | | |

-------------

It's O's turn.

Enter row (0-2): 2

Enter column (0-2): 2

-------------

| | | X |

-------------

| | O | X |

-------------

| | | O |

-------------

It's X's turn.

Enter row (0-2): 1

Enter column (0-2): 0

-------------

| | | X |

-------------

| X | O | X |

-------------

| | | O |

-------------

It's O's turn.

Enter row (0-2): 0

Enter column (0-2): 0

-------------

| O | | X |

-------------

| X | O | X |

-------------

| | | O |

-------------

O wins!

**3. Aim: Implement Monkey-Banana Problem using prolog**

**Description:**

The Monkey and Banana Problem is a classic problem in the field of artificial intelligence. It is an

example of a planning problem that involves an agent (in this case, a monkey) trying to reach a goal (in this case, a banana) in a given environment, while dealing with obstacles and constraints.

The problem is as follows:

A monkey is in a room that contains a banana, a chair, and a box. The monkey wants to reach the banana, which is hanging from the ceiling. However, the monkey cannot reach the banana directly. Instead, the

monkey must use the chair and the box strategically to reach its goal.

The monkey can perform the following actions:

-Move left or right in the room.

-Climb on the chair or get off the chair.

-Push the chair to the left or to the right.

-Climb on the box or get off the box.

-Push the box to the left or to the right.

-Take the banana once it is within reach.

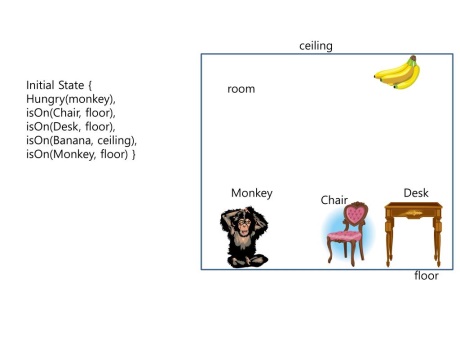


However, there are some constraints:

1.The monkey can only push the chair or the box if it is next to them.

2.The monkey cannot climb on the chair or the box if they are not on the ground.

3.The monkey can only take the banana if it is on the box.

The goal is for the monkey to create a sequence of actions that will allow it to reach the banana and take it. 

To solve this problem using artificial intelligence techniques, one approach is to use a search algorithm,

such as the A\* search algorithm. The problem can be modeled as a search problem, where the states

represent the configurations of the objects in the room and the actions represent the possible moves the

monkey can make.

The search algorithm will explore different sequences of actions to find the optimal path from the initial state (monkey on the ground, banana hanging) to the goal state (monkey on the box, banana taken). The algorithm will take into account the constraints and use heuristics to guide the search towards the most

promising paths.

By applying a search algorithm to the Monkey and Banana Problem, an AI agent can determine the

sequence of actions that the monkey needs to take in order to reach the banana and successfully solve the problem.

**Prolog code:**

on(floor,monkey).

on(floor,box).

in(room,monkey).

in(room,box).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,box).

push(monkey,box):-

strong(monkey).

under(banana,box):-

push(monkey,box).

canreach(banana,monkey):-

at(floor,banana);

at(ceiling,banana),

under(banana,box),

climb(monkey,box).

canget(banana,monkey):-

canreach(banana,monkey),

grasp(monkey).

**Output:**

?- ['D:/mb.pl'].

true.

?- trace.

true.

[trace] ?- canget(banana,monkey).

Call: (10) canget(banana, monkey) ? creep

Call: (11) canreach(banana, monkey) ? creep

Call: (12) at(floor, banana) ? creep

Fail: (12) at(floor, banana) ? creep

Redo: (11) canreach(banana, monkey) ? creep

Call: (12) at(ceiling, banana) ? creep

Exit: (12) at(ceiling, banana) ? creep

Call: (12) under(banana, box) ? creep

Call: (13) push(monkey, box) ? creep

Call: (14) strong(monkey) ? creep

Exit: (14) strong(monkey) ? creep

Exit: (13) push(monkey, box) ? creep

Exit: (12) under(banana, box) ? creep

Call: (12) climb(monkey, box) ? creep

Exit: (12) climb(monkey, box) ? creep

Exit: (11) canreach(banana, monkey) ? creep

Call: (11) grasp(monkey) ? creep

Exit: (11) grasp(monkey) ? creep

Exit: (10) canget(banana, monkey) ? creep

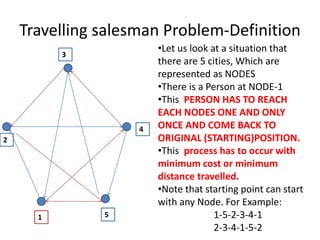
true.

**4. Aim: Implementation of TSP using heuristic approach using Java/LISP/Prolog**

**Description:**

The Traveling Salesman Problem (TSP) is a classic optimization problem in the field of Artificial Intelligence (AI) and operations research. It involves finding the shortest possible route that a salesman can take to visit a set of cities exactly once and return to the starting city. The heuristic approach to solving the TSP involves using approximation algorithms that provide reasonably good solutions, although not necessarily optimal.

In the heuristic approach, the emphasis is on finding a good solution in a reasonable amount of time, rather than guaranteeing the optimal solution. Heuristic algorithms for the TSP make use of various strategies to iteratively build a tour that gradually improves the objective function (total distance). These strategies include selecting the nearest unvisited city, choosing the city with the smallest increase in distance, or using a combination of heuristics.



One popular heuristic approach for the TSP is the Nearest Neighbor algorithm, where the salesman starts at a random city and repeatedly visits the nearest unvisited city until all cities have been visited. This approach provides a fast and simple solution but does not guarantee the optimal route.

**Prolog Code:**

edge(a, b, 3).

edge(a, c, 4).

edge(a, d, 2).

edge(a, e, 7).

edge(b, c, 4).

edge(b, d, 6).

edge(b, e, 3).

edge(c, d, 5).

edge(c, e, 8).

edge(d, e, 6).

edge(b, a, 3).

edge(c, a, 4).

edge(d, a, 2).

edge(e, a, 7).

edge(c, b, 4).

edge(d, b, 6).

edge(e, b, 3).

edge(d, c, 5).

edge(e, c, 8).

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:**

['D:/tsp.pl'].

true.

?- shortest\_path(Path).

Path = 20-[a, h, d, e, b, c, a].

yes.