**Experiment No. 1**

**DFS Algorithm Program:**

graph1 **=** {

    'A'**:** set(['B'**,** 'C'])**,**

    'B'**:** set(['A'**,** 'D'**,** 'E'])**,**

    'C'**:** set(['A'**,** 'F'])**,**

    'D'**:** set(['B'])**,**

    'E'**:** set(['B'**,** 'F'])**,**

    'F'**:** set(['C'**,** 'E'])

}

**def** dfs(**graph,** **node,** **visited**)**:**

**if** node **not** **in** visited**:**

        visited**.**append(node)

**for** n **in** graph[node]**:**

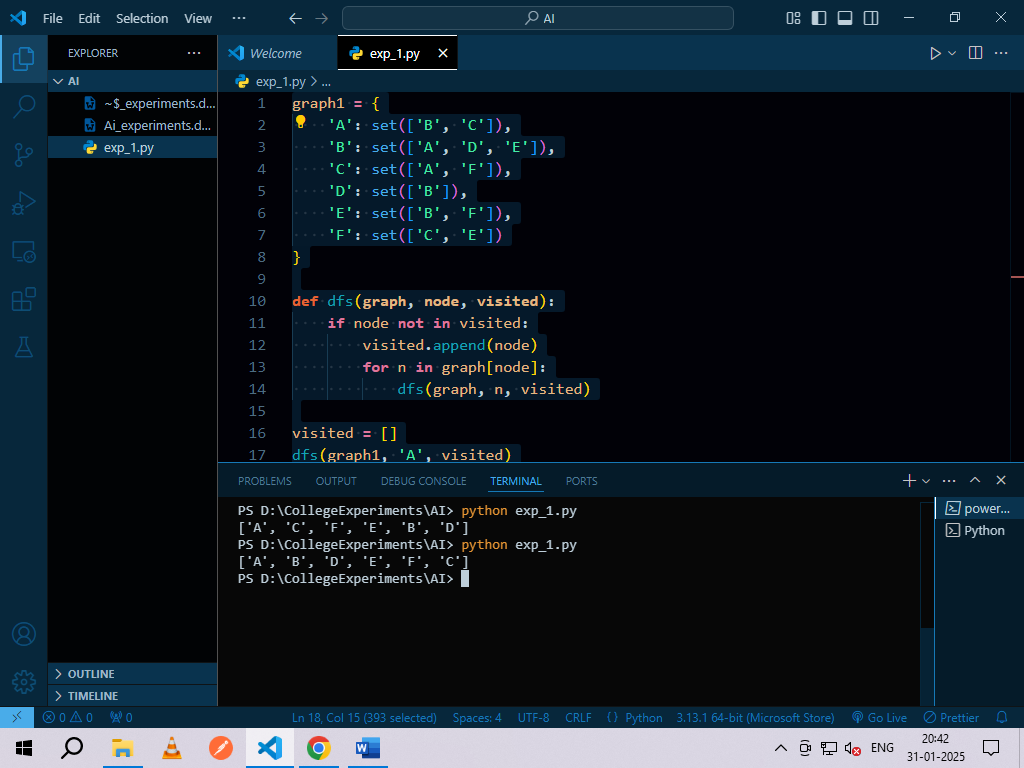
            dfs(graph**,** n**,** visited)

visited **=** []

dfs(graph1**,** 'A'**,** visited)

print(visited)

**Output:**

****

**Experiment No. 2**

**N-Queen Problem:**

class QueenChessBoard:

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

# Board has dimensions size x size

self.size = size

# Columns[r] is a number c if a queen is placed at row r and column c.

# Columns[r] is out of range if no queen is placed in row r.

# Thus, after all queens are placed, they will be at positions

# (columns[0], 0), (columns[1], 1), ... (columns[size - 1], size - 1)

self.columns = []

def place\_in\_next\_row(self, column):

self.columns.append(column)

def remove\_in\_current\_row(self):

return self.columns.pop()

def is\_this\_column\_safe\_in\_next\_row(self, column):

# Index of next row

row = len(self.columns)

# Check column

for queen\_column in self.columns:

if column == queen\_column:

return False

# Check diagonal

for queen\_row, queen\_column in enumerate(self.columns):

if queen\_column - queen\_row == column - row:

return False

# Check other diagonal

for queen\_row, queen\_column in enumerate(self.columns):

if ((self.size - queen\_column) - queen\_row

== (self.size - column) - row):

return False

return True

def display(self):

for row in range(self.size):

for column in range(self.size):

if column == self.columns[row]:

print('Q', end=' ')

else:

print('.', end=' ')

print()

def solve\_queen(size):

"""Display a chessboard for each possible configuration of placing n

queens on an n x n chessboard and print the number of such

configurations."""

board = QueenChessBoard(size)

number\_of\_solutions = 0

row = 0

column = 0

# Iterate over rows of the board

while True:

# Place queen in the next row

while column < size:

if board.is\_this\_column\_safe\_in\_next\_row(column):

board.place\_in\_next\_row(column)

row += 1

column = 0

break

else:

column += 1

# If could not find a column to place in or if the board is full

if column == size or row == size:

# If the board is full, we have a solution

if row == size:

board.display()

print()

number\_of\_solutions += 1

# Backtrack to find more solutions

board.remove\_in\_current\_row()

row -= 1

# Now backtrack

try:

prev\_column = board.remove\_in\_current\_row()

except IndexError:

# All queens removed; no more possible configurations

break

# Try the previous row again

row -= 1

# Start checking at column = (1 + value of column in the previous row)

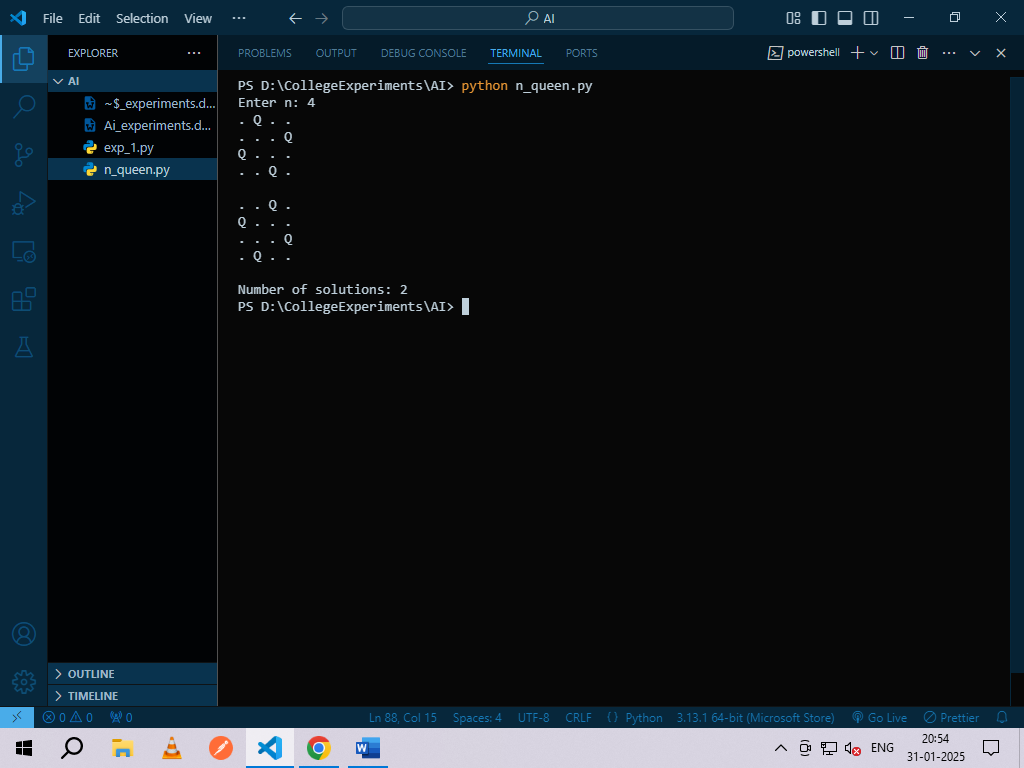
column = 1 + prev\_column

print('Number of solutions:', number\_of\_solutions)

n = int(input('Enter n: '))

solve\_queen(n)

**Output:**



**Experiment No. 3**

**Alpha Beta Search:**

tree **=** [[[5**,** 1**,** 2]**,** [8**,** **-**8**,** **-**9]]**,** [[9**,** 4**,** 5]**,** [**-**3**,** 4**,** 3]]]

root **=** 0

pruned **=** 0

**def** children(**branch,** **depth,** **alpha,** **beta**)**:**

*global* tree

*global* root

*global* pruned

    i **=** 0

**for** child **in** branch**:**

**if** type(child) **is** *list***:**

            (nalpha**,** nbeta) **=** children(child**,** depth **+** 1**,** alpha**,** beta)

**if** depth **%** 2 **==** 1**:**

                beta **=** nalpha **if** nalpha **<** beta **else** beta

**else:**

                alpha **=** nbeta **if** nbeta **>** alpha **else** alpha

            branch[i] **=** alpha **if** depth **%** 2 **==** 0 **else** beta

            i **+=** 1

**else:**

**if** depth **%** 2 **==** 0 **and** alpha **<** child**:**

                alpha **=** child

**if** depth **%** 2 **==** 1 **and** beta **>** child**:**

                beta **=** child

**if** alpha **>=** beta**:**

                pruned **+=** 1

**break**

**if** depth **==** root**:**

        tree **=** alpha **if** root **==** 0 **else** beta

**return** (alpha**,** beta)

**def** alphabeta(**in\_tree=**tree**,** **start=**root**,** **upper=-**15**,** **lower=**15)**:**

*global* tree

*global* root

*global* pruned

    tree **=** in\_tree

    root **=** start

    pruned **=** 0

    alpha**,** beta **=** children(tree**,** root**,** lower**,** upper)

    print("(alpha, beta):"**,** (alpha**,** beta))

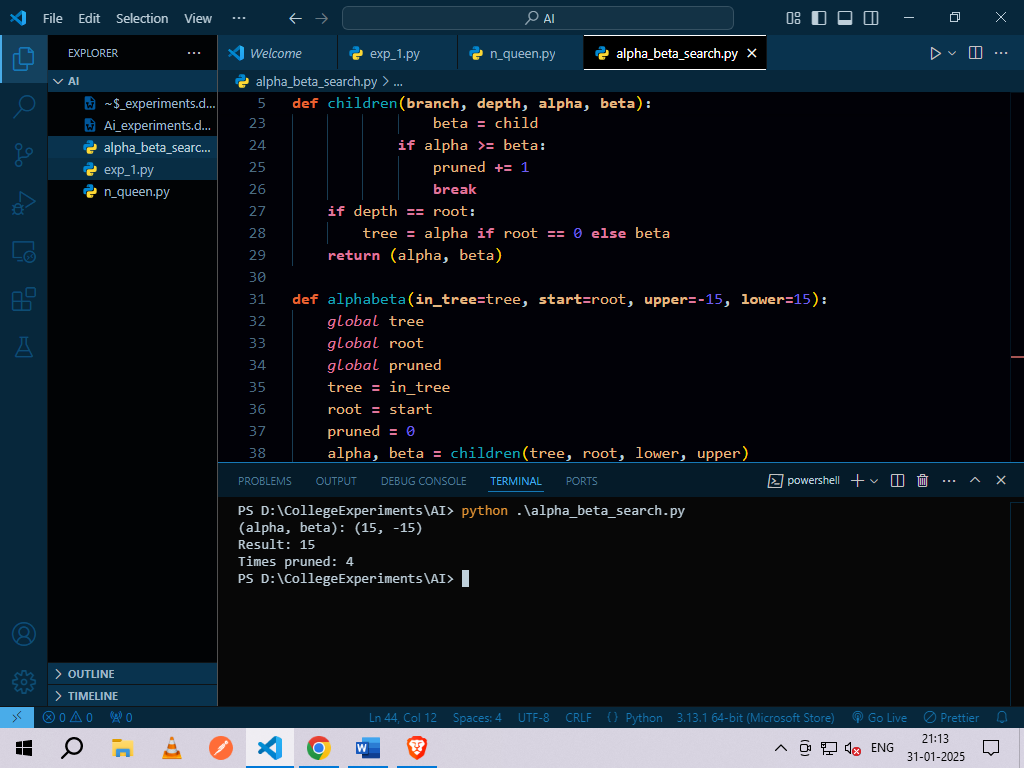
    print("Result:"**,** tree)

    print("Times pruned:"**,** pruned)

*# Run the algorithm*

alphabeta()

**Output:**

****

**Experiment No. 4**

**A\* Algorithm:**

**from** simpleai**.**search **import** SearchProblem**,** astar

GOAL **=** 'HELLO WORLD'

class **HelloProblem**(*SearchProblem*)**:**

**def** actions(*self***,** **state**)**:**

**if** len(state) **<** len(GOAL)**:**

**return** list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')  *# Possible actions (characters)*

**else:**

**return** []  *# No more actions once the goal length is reached*

**def** result(*self***,** **state,** **action**)**:**

**return** state **+** action  *# Append the action (single character) to the state*

**def** is\_goal(*self***,** **state**)**:**

**return** state **==** GOAL  *# Goal is reached when state matches the GOAL string*

**def** heuristic(*self***,** **state**)**:**

*# Calculate the heuristic: number of wrong characters + missing characters*

        wrong **=** sum([1 **if** state[i] **!=** GOAL[i] **else** 0

**for** i **in** range(len(state))])  *# Count wrong characters*

        missing **=** len(GOAL) **-** len(state)  *# Count missing characters (if any)*

**return** wrong **+** missing  *# Total wrong + missing letters*

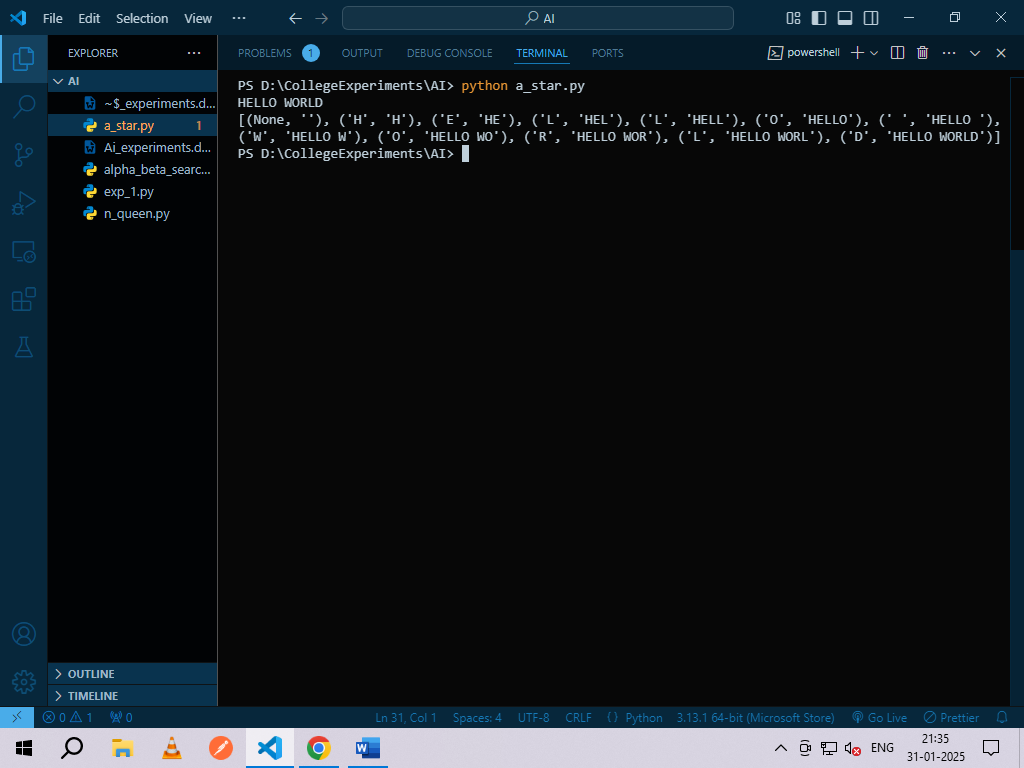
problem **=** HelloProblem(**initial\_state=**'')

result **=** astar(problem)

print(result**.**state)

print(result**.**path())

**Output:**



**Experiment No. 5**

**TIC – TAC –TOE game using Min-Max Algorithm:**

**import** time

board **=** [' '**,** ' '**,** ' '**,** ' '**,** ' '**,** ' '**,** ' '**,** ' '**,** ' '**,** ' ']

player **=** 1

*########win Flags##########*

Win **=** 1

Draw **=** **-**1

Running **=** 0

Stop **=** 1

*###########################*

Game **=** Running

Mark **=** 'X'

*# This Function Draws Game Board*

**def** DrawBoard()**:**

    print(" %c | %c | %c " **%** (board[1]**,** board[2]**,** board[3]))

    print("\_\_\_|\_\_\_|\_\_\_")

    print(" %c | %c | %c " **%** (board[4]**,** board[5]**,** board[6]))

    print("\_\_\_|\_\_\_|\_\_\_")

    print(" %c | %c | %c " **%** (board[7]**,** board[8]**,** board[9]))

    print("   |   |   ")

*# This Function Checks position is empty or not*

**def** CheckPosition(**x**)**:**

**if** board[x] **==** ' '**:**

**return** True

**else:**

**return** False

*# This Function Checks player has won or not*

**def** CheckWin()**:**

*global* Game

*# Horizontal winning condition*

**if** (board[1] **==** board[2] **and** board[2] **==** board[3] **and** board[1] **!=** ' ')**:**

        Game **=** Win

**elif** (board[4] **==** board[5] **and** board[5] **==** board[6] **and** board[4] **!=** ' ')**:**

        Game **=** Win

**elif** (board[7] **==** board[8] **and** board[8] **==** board[9] **and** board[7] **!=** ' ')**:**

        Game **=** Win

*# Vertical Winning Condition*

**elif** (board[1] **==** board[4] **and** board[4] **==** board[7] **and** board[1] **!=** ' ')**:**

        Game **=** Win

**elif** (board[2] **==** board[5] **and** board[5] **==** board[8] **and** board[2] **!=** ' ')**:**

        Game **=** Win

**elif** (board[3] **==** board[6] **and** board[6] **==** board[9] **and** board[3] **!=** ' ')**:**

        Game **=** Win

*# Diagonal Winning Condition*

**elif** (board[1] **==** board[5] **and** board[5] **==** board[9] **and** board[5] **!=** ' ')**:**

        Game **=** Win

**elif** (board[3] **==** board[5] **and** board[5] **==** board[7] **and** board[5] **!=** ' ')**:**

        Game **=** Win

*# Match Tie or Draw Condition*

**elif** (board[1] **!=** ' ' **and** board[2] **!=** ' ' **and** board[3] **!=** ' ' **and**

          board[4] **!=** ' ' **and** board[5] **!=** ' ' **and** board[6] **!=** ' ' **and**

          board[7] **!=** ' ' **and** board[8] **!=** ' ' **and** board[9] **!=** ' ')**:**

        Game **=** Draw

**else:**

        Game **=** Running

print("Tic-Tac-Toe Game")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print()

print("Please Wait...")

time**.**sleep(1)

**while** Game **==** Running**:**

    DrawBoard()  *# Draw the board without clearing the screen*

**if** player **%** 2 **!=** 0**:**

        print("Player 1's chance")

        Mark **=** 'X'

**else:**

        print("Player 2's chance")

        Mark **=** 'O'

    choice **=** int(input("Enter the position between [1-9] where you want to mark: "))

**if** CheckPosition(choice)**:**

        board[choice] **=** Mark

        player **+=** 1

        CheckWin()

**else:**

        print("Position already occupied. Try again.")

        time**.**sleep(1)  *# Pause to let the player read the message*

DrawBoard()  *# Draw the final board*

**if** Game **==** Draw**:**

    print("Game Draw")

**elif** Game **==** Win**:**

    player **-=** 1

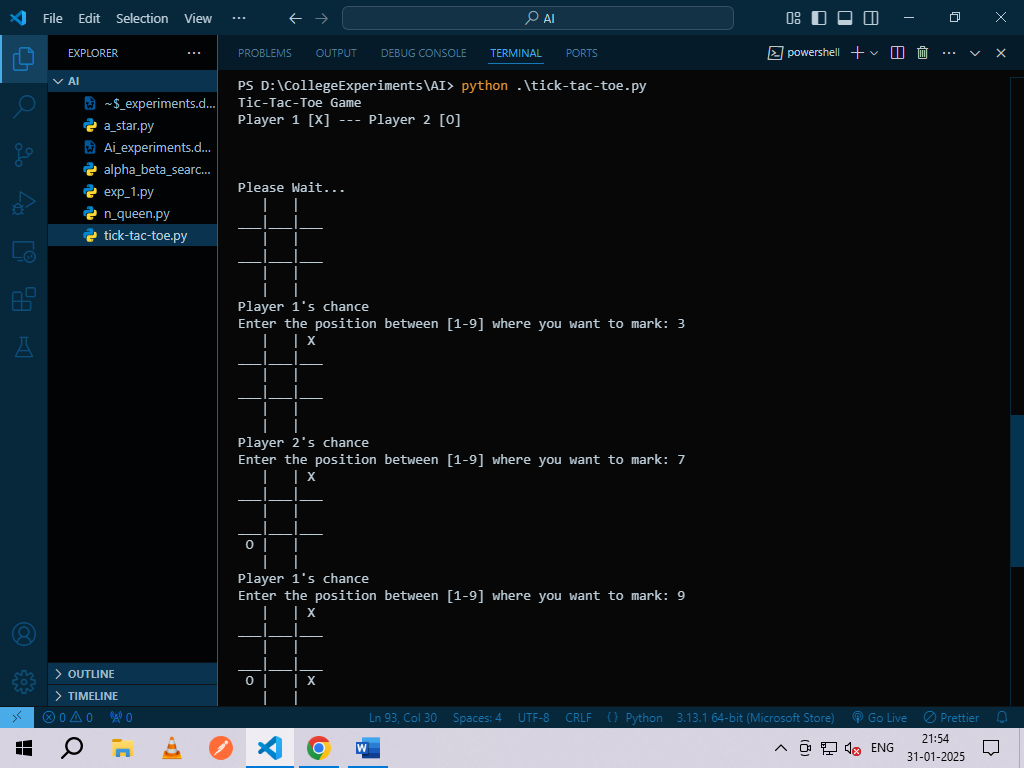
**if** player **%** 2 **!=** 0**:**

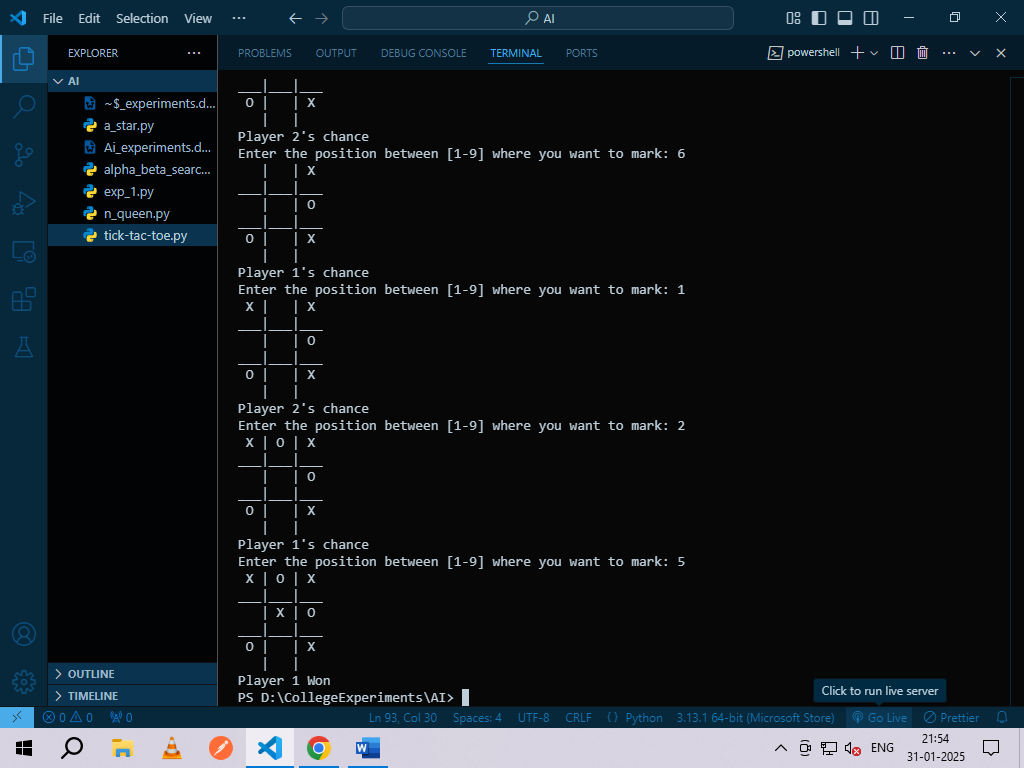
        print("Player 1 Won")

**else:**

        print("Player 2 Won")

**Output:**

****



**Experiment NO. 6**

**Number Puzzle Game:**

**from** \_\_future\_\_ **import** print\_function

**from** simpleai**.**search **import** astar**,** SearchProblem

**from** simpleai**.**search**.**viewers **import** WebViewer

GOAL **=** '''1-2-3

4-5-6

7-8-e'''

INITIAL **=** '''4-1-2

7-e-3

8-5-6'''

**def** list\_to\_string(**list\_**)**:**

**return** '\n'**.**join(['-'**.**join(row) **for** row **in** list\_])

**def** string\_to\_list(**string\_**)**:**

**return** [row**.**split('-') **for** row **in** string\_**.**split('\n')]

**def** find\_location(**rows,** **element\_to\_find**)**:**

    '''Find the location of a piece in the puzzle.

    Returns a tuple: row, column'''

**for** ir**,** row **in** enumerate(rows)**:**

**for** ic**,** element **in** enumerate(row)**:**

**if** element **==** element\_to\_find**:**

**return** ir**,** ic

*# Create a cache for the goal position of each piece*

goal\_positions **=** {}

rows\_goal **=** string\_to\_list(GOAL)

**for** number **in** '12345678e'**:**

    goal\_positions[number] **=** find\_location(rows\_goal**,** number)

class **EightPuzzleProblem**(*SearchProblem*)**:**

**def** actions(*self***,** **state**)**:**

        '''Returns a list of the pieces we can move to the empty space.'''

        rows **=** string\_to\_list(state)

        row\_e**,** col\_e **=** find\_location(rows**,** 'e')

        actions **=** []

**if** row\_e **>** 0**:**

            actions**.**append(rows[row\_e **-** 1][col\_e])  *# Move the piece above 'e'*

**if** row\_e **<** 2**:**

            actions**.**append(rows[row\_e **+** 1][col\_e])  *# Move the piece below 'e'*

**if** col\_e **>** 0**:**

            actions**.**append(rows[row\_e][col\_e **-** 1])  *# Move the piece to the left of 'e'*

**if** col\_e **<** 2**:**

            actions**.**append(rows[row\_e][col\_e **+** 1])  *# Move the piece to the right of 'e'*

**return** actions

**def** result(*self***,** **state,** **action**)**:**

        '''Return the resulting state after moving a piece to the empty space.'''

        rows **=** string\_to\_list(state)

        row\_e**,** col\_e **=** find\_location(rows**,** 'e')

        row\_n**,** col\_n **=** find\_location(rows**,** action)

*# Swap the empty space and the piece*

        rows[row\_e][col\_e]**,** rows[row\_n][col\_n] **=** rows[row\_n][col\_n]**,** rows[row\_e][col\_e]

**return** list\_to\_string(rows)

**def** is\_goal(*self***,** **state**)**:**

        '''Returns true if a state is the goal state.'''

**return** state **==** GOAL

**def** cost(*self***,** **state1,** **action,** **state2**)**:**

        '''Returns the cost of performing an action. No useful on this problem,

        but needed.'''

**return** 1

**def** heuristic(*self***,** **state**)**:**

        '''Returns an \*estimation\* of the distance from a state to the goal.

        We are using the Manhattan distance.'''

        rows **=** string\_to\_list(state)

        distance **=** 0

**for** number **in** '12345678e'**:**

            row\_n**,** col\_n **=** find\_location(rows**,** number)

            row\_n\_goal**,** col\_n\_goal **=** goal\_positions[number]

            distance **+=** abs(row\_n **-** row\_n\_goal) **+** abs(col\_n **-** col\_n\_goal)

**return** distance

result **=** astar(EightPuzzleProblem(INITIAL))

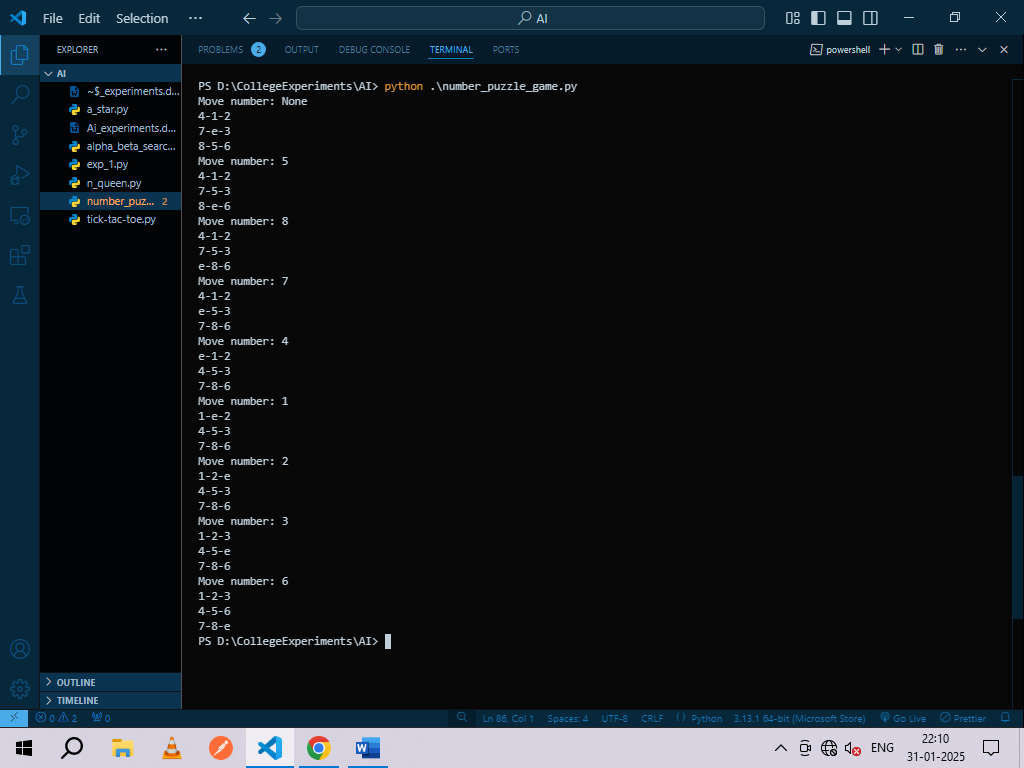
*# Print the result path*

**for** action**,** state **in** result**.**path()**:**

    print('Move number:'**,** action)

    print(state)

**Output:**

****

**Experiment No. 7**

**Shuffle Deck of Cards:**

*# Import random module since we will be shuffling*

**import** random

*# Create lists to hold card faces, suits, and the deck*

cardfaces **=** []

suits **=** ["Hearts"**,** "Diamonds"**,** "Clubs"**,** "Spades"]

royals **=** ["J"**,** "Q"**,** "K"**,** "A"]

deck **=** []

*# Add number faces (2-10) to the cardfaces list*

**for** i **in** range(2**,** 11)**:**

    cardfaces**.**append(str(i))  *# Adds numbers 2-10 as strings*

*# Add royal faces (J, Q, K, A) to the cardfaces list*

**for** j **in** range(4)**:**

    cardfaces**.**append(royals[j])

*# Create the full deck by combining cardfaces with suits*

**for** suit **in** suits**:**

**for** face **in** cardfaces**:**

        card **=** face **+** " of " **+** suit  *# Format each card as 'face of suit'*

        deck**.**append(card)  *# Add each card to the deck*

*# Shuffle the deck*

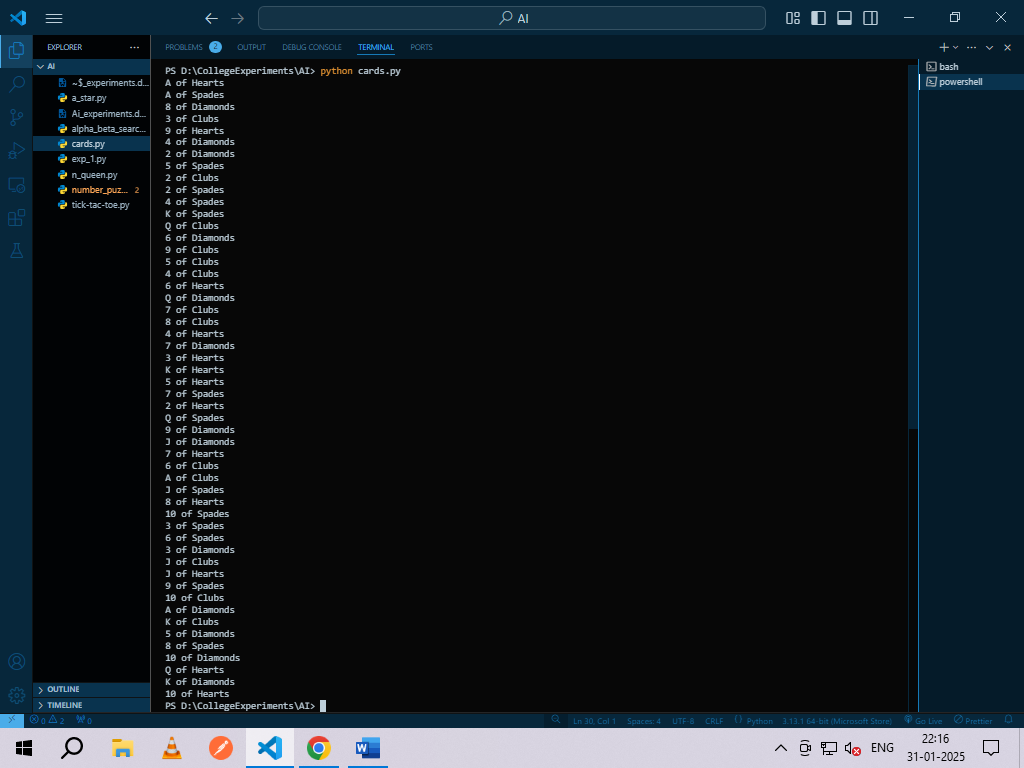
random**.**shuffle(deck)

*# Print all the shuffled cards*

**for** card **in** deck**:**

    print(card)

**Output:**

****

**Drawing 5 Cards:**

# Import required modules

import itertools

import random

# Create the deck of cards

deck = list(itertools.product(range(1, 14), ['Spade', 'Heart', 'Diamond', 'Club']))

# Shuffle the deck

random.shuffle(deck)

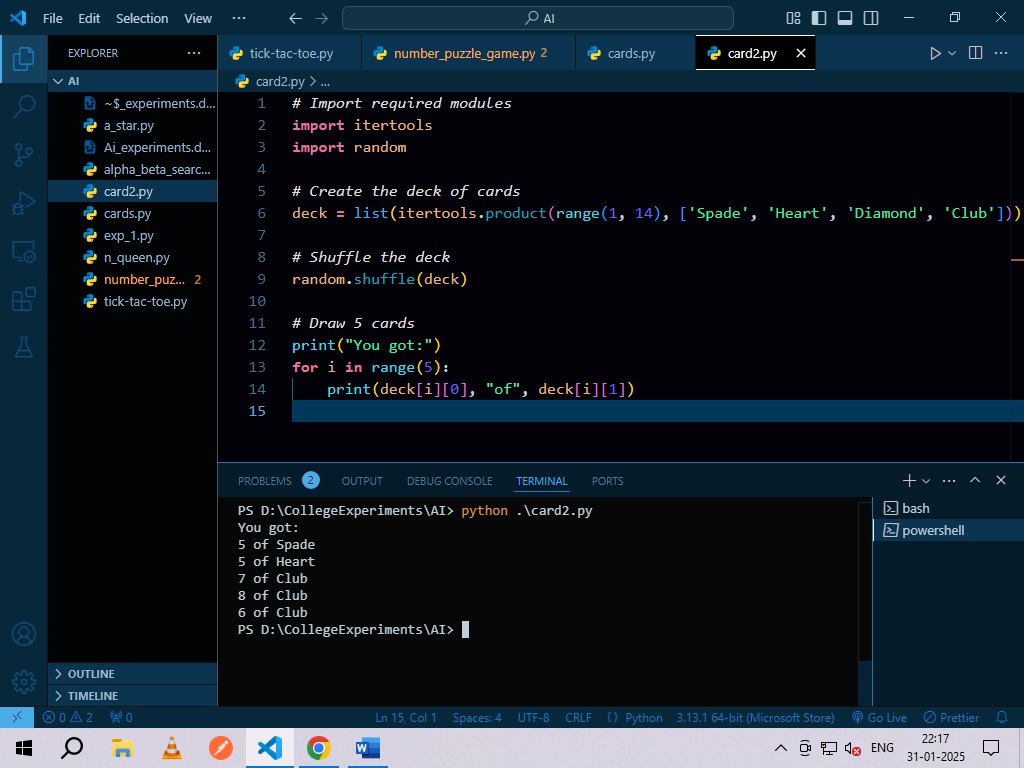
# Draw 5 cards

print("You got:")

for i in range(5):

print(deck[i][0], "of", deck[i][1])

**Output:**

****

**Experiment No. 8**

**Block of World Problem:**

from \_\_future\_\_ import print\_function

from simpleai.search import CspProblem, backtrack, min\_conflicts

from simpleai.search.csp import MOST\_CONSTRAINED\_VARIABLE, HIGHEST\_DEGREE\_VARIABLE, LEAST\_CONSTRAINING\_VALUE

variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')

domains = dict((v, ['red', 'green', 'blue']) for v in variables)

def const\_different(variables, values):

return values[0] != values[1] # Expect the value of the neighbors to be different

constraints = [

(('WA', 'NT'), const\_different),

(('WA', 'SA'), const\_different),

(('SA', 'NT'), const\_different),

(('SA', 'Q'), const\_different),

(('NT', 'Q'), const\_different),

(('SA', 'NSW'), const\_different),

(('Q', 'NSW'), const\_different),

(('SA', 'V'), const\_different),

(('NSW', 'V'), const\_different),

]

my\_problem = CspProblem(variables, domains, constraints)

print(backtrack(my\_problem)) # Using default backtrack

print(backtrack(my\_problem, variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE)) # Using most constrained variable heuristic

print(backtrack(my\_problem, variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE)) # Using highest degree variable heuristic

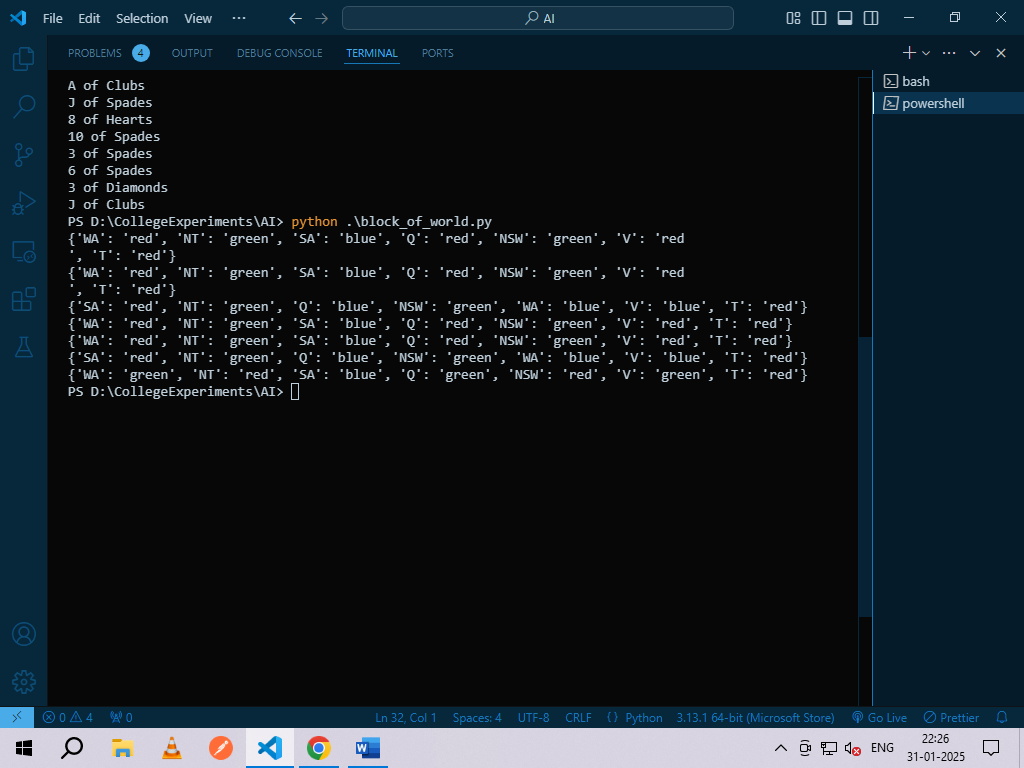
print(backtrack(my\_problem, value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) # Using least constraining value heuristic

print(backtrack(my\_problem, variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) # Both heuristics

print(backtrack(my\_problem, variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE)) # Both heuristics

print(min\_conflicts(my\_problem)) # Using the min conflicts heuristic

**Output:**

****