ARTIFICIAL INTELLIGENCE

LAB MANUAL

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AUUP, NOIDA

B.TECH – COMPUTER SCIENCE AND ENGINEERING

SEMESTER – 6

COURSE CODE – CSE401

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMITY UNIVERSITY, NOIDA, UTTARPRADESH

Name: Akhil Agrawal

Enrollment No: A2305222080

Section: 6CSE2X

Submitted To: Dr. Anil Sharma

INDEX

S.No	Experiments	Date of experiment	Date of Submission	Remarks
1.	1.Write a Program to Check Prime Number 2. Write a Program to Print the Fibonacci sequence. 3. Write a Program to Find the Factorial of a Number. 4. Write a program to reverse digits of a number. 5. Write Program in python to swap two numbers.			
2.	Write a program to implement the Tic-Tac-Toe problem.			
3.	Write a program to Implement a single Player game.			
4.	Write a program to implement a water jug problem in python.			
5.	Implement a Brute force solution to the Knapsack problem in Python.			
6.	Write a program to implement A* algorithm in python			
7.	Write a program to implement BFS for water jug problem using Python			
8.	Write a program to implement DFS using Python.			

Experiment:1

Aim: 1.Write a Program to Check Prime Number

Language Used: Python

Theory: A prime number is a number greater than 1 that has no divisors other than 1 and itself. This program checks if a given number is prime by iterating from 2 to the square root of the number (as factors repeat beyond this range). If any number divides the input without a remainder, it is not prime. Otherwise, it is declared a prime number.

Source Code:

```
num = int(input("Enter a number: "))

if num > 1:
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            print(f"{num} is not a prime number")
            break
    else:
        print(f"{num} is a prime number")
else:
    print(f"{num} is not a prime number")
```

Output:

```
Enter a number: 5
5 is a prime number
```

2. Write a Program to Print the Fibonacci sequence

Language Used: Python

Theory: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1. This program generates the Fibonacci sequence up to n terms by initializing the first two numbers (a and b) and iteratively updating them while printing the current term. The process continues for the specified number of terms.

Source Code:

```
n = int(input("Enter the number of terms: "))

a, b = 0, 1

for _ in range(n):
    print(a, end=" ")
    a, b = b, a + b
```

Output:

```
Enter the number of terms: 5
0 1 1 2 3
```

3. Write a Program to Find the Factorial of a Number

Language Used: Python

Theory: The factorial of a number is the product of all positive integers from 1 to that number (denoted as n!). This program calculates the factorial by initializing a variable to 1 and multiplying it by each number from 1 to the input value using a loop. The final result represents the factorial of the given number.

Source Code:

```
num = int(input("Enter a number: "))

factorial = 1


for i in range(1, num + 1):

    factorial *= i


print(f"The factorial of {num} is {factorial}")
```

Output:

```
Enter a number: 12
The factorial of 12 is 479001600
```

4. Write a program to reverse digits of a number

Language Used: Python

Theory: Reversing the digits of a number involves rearranging its digits in reverse order. This program extracts each digit of the input number using the modulo operator (% 10) and appends it to the reversed number by multiplying the current reversed number by 10 and adding the digit. The original number is reduced by dividing it by 10 in each iteration. The process continues until all digits are reversed.

Source Code:

```
num = int(input("Enter a number: "))

reversed_num = 0


while num > 0:

    reversed_num = reversed_num * 10 + num % 10

    num //= 10
```

print(f"Reversed number: {reversed_num}")

Output:

```
Enter a number: 123574
Reversed number: 475321
```

5. Write Program in python to swap two numbers

Language Used: Python

Theory: Swapping two numbers means exchanging their values. This program achieves the swap by using a temporary variable. The value of the first number (a) is stored in temp, then a is assigned the value of b, and finally b takes the value of temp. This way, the two numbers are swapped without losing any values.

Source Code:

```
a = int(input("Enter the first number: "))

b = int(input("Enter the second number: "))

temp = a

a = b

b = temp

print(f"After swapping: a = {a}, b = {b}")
```

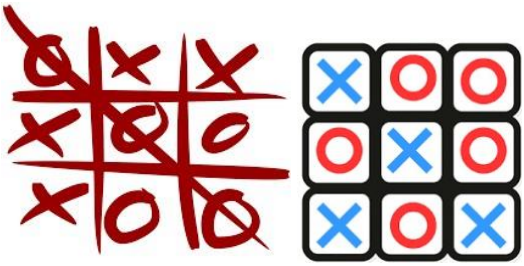
Output:

```
Enter the first number: 45
Enter the second number: 34
After swapping: a = 34, b = 45
```

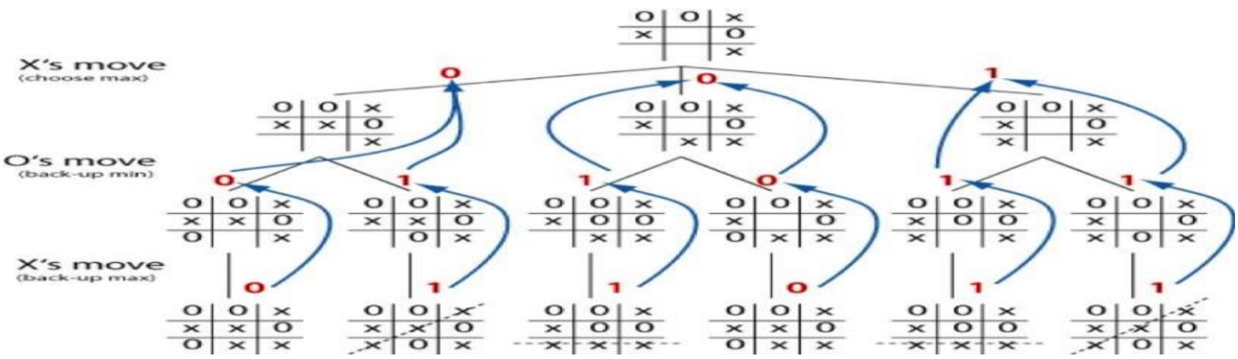
EXPERIMENT-2

Objective: Write a program to implement the Tic-Tac-Toe game problem

Software Used: Python



Theory: Tic-tac-toe is a classic two-player game where participants take turns marking spaces in a 3×3 grid. The objective is to be the first to align three of one's marks horizontally, vertically, or diagonally. With optimal strategies, the game inevitably ends in a draw, highlighting its simplicity and deterministic nature. This simplicity makes tic-tac-toe a favourite among young children, who often approach it with fresh enthusiasm, unaware of the inevitability of a draw under perfect play. The game also serves as an excellent tool for teaching the fundamentals of strategic thinking, sportsmanship, and basic concepts in artificial intelligence, such as game tree exploration and state space analysis. In computational terms, tic-tac-toe has 765 distinct board states and 26,830 unique games when accounting for symmetrical transformations like rotations and reflections. It is a specific case of the generalized (m, n, k)-game, where players aim to align k marks on an m×n grid. As a (3, 3, 3)-game, it epitomizes simplicity in design and outcome, demonstrating that proper play always leads to a draw, making it a perfect example of a futile yet educational pastime.



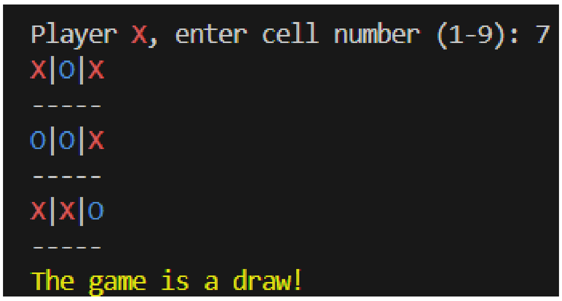
Code:

```
class TicTacToe:

    def __init__(self):  # Fixed double underscore in `__init__`

        self.board = [" "] * 9

        self.current_player = "\u001b[31mX\u001b[0m"  # Red-colored X


    def display_board(self):

        for row in [self.board[i:i + 3] for i in range(0, 9, 3)]:

            print("|".join(row))

            print("-" * 5)


    def make_move(self, position):

        if self.board[position] == " ":

            self.board[position] = self.current_player

            if self.check_winner():

                self.display_board()  # Display the final board

                print(f"\u001b[32mPlayer {self.current_player} wins!\u001b[0m")

                return True

            if self.check_draw():

                self.display_board()  # Display the final board

                print("\u001b[33mThe game is a draw!\u001b[0m")

                return True

            # Switch player

                    self.current_player = "\u001b[34mO\u001b[0m" if self.current_player ==
"\u001b[31mX\u001b[0m" else "\u001b[31mX\u001b[0m"
```

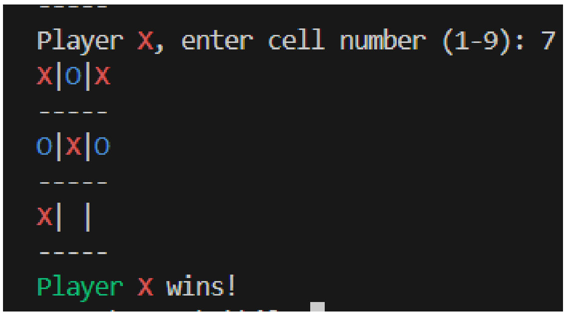
```
        else:
            print("\u001b[33mCell already taken. Try again.\u001b[0m")
        return False

    def check_winner(self):
        win_conditions = [
            (0, 1, 2), (3, 4, 5), (6, 7, 8),  # Rows
            (0, 3, 6), (1, 4, 7), (2, 5, 8),  # Columns
            (0, 4, 8), (2, 4, 6)              # Diagonals
        ]
        for a, b, c in win_conditions:
            if self.board[a] == self.board[b] == self.board[c] != " ":
                return True
        return False

    def check_draw(self):
        return all(cell != " " for cell in self.board)

def play_tic_tac_toe():
    game = TicTacToe()
    while True:
        game.display_board()
        try:
            # Adjust user input (1-9) to match index (0-8)
            move = int(input(f"Player {game.current_player}, enter cell number (1-9): ")) - 1
            if move < 0 or move >= 9:
```

```
            print("\u001b[33mInvalid input. Please enter a number between 1 and 9.\u001b[0m")
            continue
        if game.make_move(move):
            break
    except ValueError:
        print("\u001b[33mInvalid input. Please enter a valid number.\u001b[0m")
play_tic_tac_toe()
```

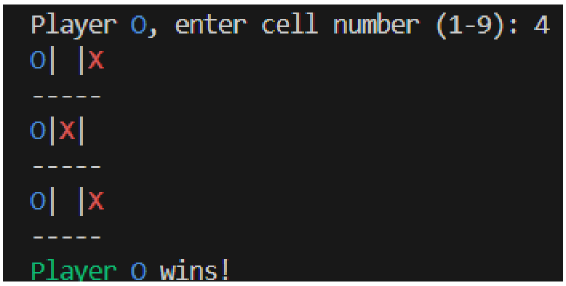
OUTPUT



```
Player X, enter cell number (1-9): 7
X|O|X
-----
O|O|X
-----
X|X|O
-----
The game is a draw!
```



```
-----
Player X, enter cell number (1-9): 7
X|O|X
-----
O|X|O
-----
X| |
-----
Player X wins!
```



```
Player O, enter cell number (1-9): 4
O| |X
-----
O|X|
-----
O| |X
-----
Player O wins!
```

Experiment:3

Aim: Write a program to implement a Single Player Game

Software Used: Python

Theory: N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24 and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns. E.g. 15-Puzzle will have 4 rows and 4 columns and an 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

Code:

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval
```

```
def generate_child(self):
    x, y = self.find(self.data, '_')
    val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
    children = []
    for i in val_list:
        child = self.shuffle(self.data, x, y, i[0], i[1])
        if child is not None:
            child_node = Node(child, self.level + 1, 0)
            children.append(child_node)
    return children

def shuffle(self, puz, x1, y1, x2, y2):
    if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data):
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][y1]
        temp_puz[x1][y1] = temp
        return temp_puz
    else:
        return None

def copy(self, root):
    temp = []
```

```
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self, puz, x):
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j

class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
```

```python
        return puz

    def f(self, start, goal):
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp

    def process(self):
        print("Enter the start state matrix:")
        start = self.accept()
        print("Enter the goal state matrix:")
        goal = self.accept()
        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        self.open.append(start)
        print("\nSolution Steps:\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
```

```
        print(" \\/\n")
        for i in cur.data:
            for j in i:
                print(j, end=" ")
            print("")
        if self.h(cur.data, goal) == 0:
            print("\nGoal state reached!")
            break
        for i in cur.generate_child():
            i.fval = self.f(i, goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]
        self.open.sort(key=lambda x: x.fval, reverse=False)


puz = Puzzle(3)
puz.process()
```

Output:

```
Enter the start state matrix:
1 2 3
8   4
7 6 5
Enter the goal state matrix:
2 8 1
  4 3
7 6 5

Solution Steps:


  |
  |
 \/

1 2 3
  |
 \/

1 2 3
 \/

1 2 3
8   4

1 2 3
```

```
 Traceback (most recent call last):
 1 2 3
 8   4
 7 6 5
```

Experiment:4

Aim: Write a program to implement water jug problem using Python.

Language Used: Python

Theory: Breadth-First Search (BFS) is similar to Breadth-First Traversal of a tree. However, unlike trees, graphs may contain cycles, which means we may revisit the same node multiple times. To prevent unnecessary processing, we use a Boolean visited array to mark nodes that have already been explored.

For simplicity, we assume that all vertices are reachable from the starting vertex. For instance, in the following graph, we begin traversal from vertex 2. When we reach vertex 0, we check all its adjacent vertices. Vertex 2 is also adjacent to 0, and without marking visited nodes, it would be processed again, leading to an infinite loop.

A Breadth-First Traversal of the given graph is: 2, 0, 3, 1.

Source Code:

```
capacity = (12, 8, 5)  # Maximum capacities of the three jugs

x, y, z = capacity


# To mark visited states

memory = {}

# Store solution path

ans = []


def get_all_states(state):

    a, b, c = state


    if (a == 6 and b == 6):

        ans.append(state)
```

```
    return True

  if (a, b, c) in memory:
    return False

  memory[(a, b, c)] = 1

  # Empty jug a
  if a > 0:
    # Empty a into b
    if a + b <= y:
      if get_all_states((0, a + b, c)):
        ans.append(state)
        return True
    else:
      if get_all_states((a - (y - b), y, c)):
        ans.append(state)
        return True
    # Empty a into c
    if a + c <= z:
      if get_all_states((0, b, a + c)):
        ans.append(state)
        return True
    else:
      if get_all_states((a - (z - c), b, z)):
        ans.append(state)
        return True

  # Empty jug b
  if b > 0:
    # Empty b into a
```

```
        if a + b <= x:
            if get_all_states((a + b, 0, c)):
                ans.append(state)
                return True
        else:
            if get_all_states((x, b - (x - a), c)):
                ans.append(state)
                return True
        # Empty b into c
        if b + c <= z:
            if get_all_states((a, 0, b + c)):
                ans.append(state)
                return True
        else:
            if get_all_states((a, b - (z - c), z)):
                ans.append(state)
                return True

    # Empty jug c
    if c > 0:
        # Empty c into a
        if a + c <= x:
            if get_all_states((a + c, b, 0)):
                ans.append(state)
                return True
        else:
            if get_all_states((x, b, c - (x - a))):
                ans.append(state)
                return True
        # Empty c into b
        if b + c <= y:
```

```
            if get_all_states((a, b + c, 0)):
                ans.append(state)
                return True
        else:
            if get_all_states((a, y, c - (y - b))):
                ans.append(state)
                return True

    return False

initial_state = (12, 0, 0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

Output:

```
(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

Experiment:5

Aim: Implement a Brute force solution to the Knapsack problem in Python

Language Used: Python

Theory: The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Source Code:

```
class Bounty:
    def __init__(self, value, weight, volume):
        self.value = value
        self.weight = weight
        self.volume = volume

# Defining bounties
panacea = Bounty(4000,  0.3, 0.025)
ichor   = Bounty(2800,  0.2, 0.015)
gold    = Bounty(3500,  2.0, 0.002)
sack    = Bounty(   0, 25.0, 0.25)

best = Bounty(0, 0, 0)
current = Bounty(0, 0, 0)
best_amounts = (0, 0, 0)

# Maximum number of items that can fit
max_panacea = int(min(sack.weight // panacea.weight, sack.volume // panacea.volume)) + 1
max_ichor   = int(min(sack.weight // ichor.weight, sack.volume // ichor.volume)) + 1
max_gold    = int(min(sack.weight // gold.weight, sack.volume // gold.volume)) + 1

# Brute-force checking all combinations
for npanacea in range(max_panacea):
```

```
    for nichor in range(max_ichor):
        for ngold in range(max_gold):
            current.value = npanacea * panacea.value + nichor * ichor.value + ngold * gold.value
            current.weight = npanacea * panacea.weight + nichor * ichor.weight + ngold * gold.weight
            current.volume = npanacea * panacea.volume + nichor * ichor.volume + ngold * gold.volume

            if current.value > best.value and current.weight <= sack.weight and current.volume <= sack.volume:
                best = Bounty(current.value, current.weight, current.volume)
                best_amounts = (npanacea, nichor, ngold)

# Printing results
print("Maximum value achievable is", best.value)
print("This is achieved by carrying %d panacea, %d ichor, and %d gold" % best_amounts)
print("The weight to carry is %.1f and the volume used is %.3f" % (best.weight, best.volume))
```

Output:

```
Maximum value achievable is 80500
This is achieved by carrying 0 panacea, 15 ichor, and 11 gold
The weight to carry is 25.0 and the volume used is 0.247


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment:6

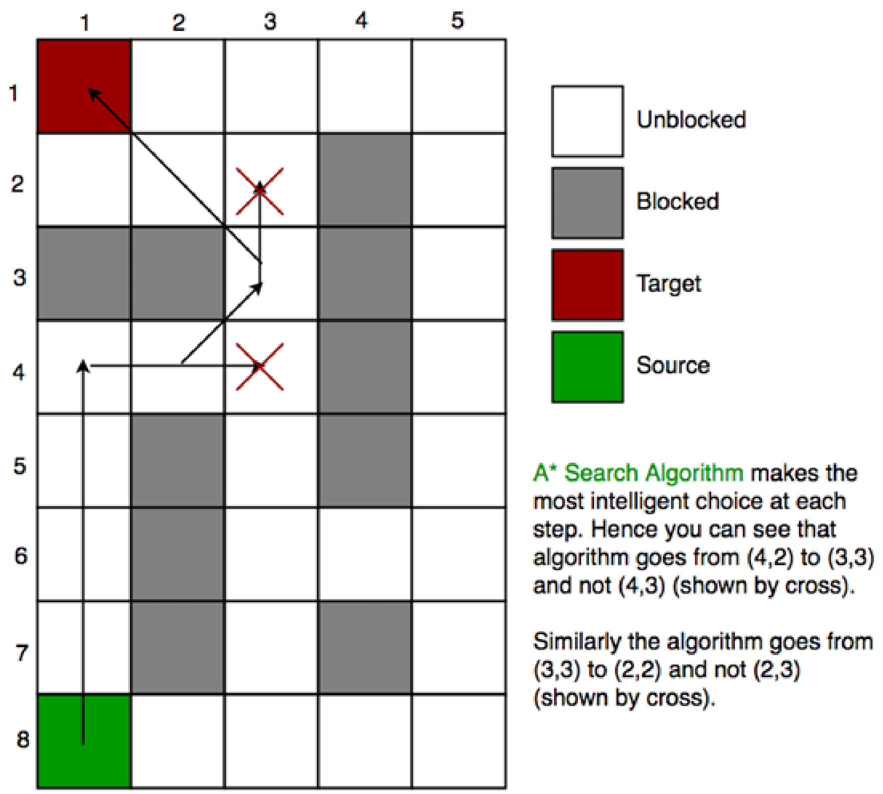
Aim: Write a program to implement A* algorithm in python

Language Used: Python

Theory: A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals. Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value 'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell. We define 'g' and 'h' as simply as possible below:

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this 'h'.



Source Code:

```
class Node:
    """A node class for A* Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
```

```
        self.position = position
        self.g = 0  # Cost from start node
        self.h = 0  # Heuristic cost to end node
        self.f = 0  # Total cost

    def __eq__(self, other):
        return self.position == other.position

def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    start_node = Node(None, start)
    end_node = Node(None, end)

    open_list = []
    closed_list = []

    open_list.append(start_node)

    while open_list:

        current_node = min(open_list, key=lambda node: node.f)
        open_list.remove(current_node)
        closed_list.append(current_node)

        if current_node == end_node:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path
```

```
        children = []

        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] +
new_position[1])

            if node_position[0] < 0 or node_position[0] >= len(maze) or node_position[1] < 0 or
node_position[1] >= len(maze[0]):
                continue

            if maze[node_position[0]][node_position[1]] != 0:
                continue

            new_node = Node(current_node, node_position)
            children.append(new_node)

        for child in children:
            if child in closed_list:
                continue

            child.g = current_node.g + 1
            child.h = (child.position[0] - end_node.position[0]) ** 2 + (child.position[1] -
end_node.position[1]) ** 2
            child.f = child.g + child.h

            if any(open_node for open_node in open_list if child == open_node and child.g >
open_node.g):
                continue

            open_list.append(child)
```

```
    return None  # No path found


def main():
    maze = [
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    ]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    if path:
        print("The required path is:")
        print(path)
    else:
        print("No valid path found.")
if __name__ == '__main__':
    main()
```

Output:

```
The required path is:
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
```

Experiment:7

Aim: Write a program to implement BFS for water jug problem using Python

Language used: Python

Theory:
The Water Jug Problem is a classic problem in artificial intelligence and search algorithms. Given two jugs with fixed capacities and an unlimited water supply, the objective is to measure a specific target amount using a sequence of valid operations: filling a jug, emptying a jug, or pouring water from one jug to another. This problem can be solved using Breadth-First Search (BFS), which systematically explores all possible states to find the shortest path to the goal state.

Source Code:

```
from collections import deque

def is_goal(state, target):
  return target in state

def bfs_water_jug(capacity_x, capacity_y, target):
  visited = set()
  queue = deque()
  queue.append((0, 0))  # Initial state: both jugs empty
  visited.add((0, 0))
  while queue:
    x, y = queue.popleft()
    print(f"({x}, {y})")
    if is_goal((x, y), target):
      print("Goal reached!")
      return
    possible_moves = set()
```

```
        # Fill jug X
        possible_moves.add((capacity_x, y))
        # Fill jug Y
        possible_moves.add((x, capacity_y))
        # Empty jug X
        possible_moves.add((0, y))
        # Empty jug Y
        possible_moves.add((x, 0))
        # Pour X -> Y
        pour_x_to_y = min(x, capacity_y - y)
        possible_moves.add((x - pour_x_to_y, y + pour_x_to_y))
        # Pour Y -> X
        pour_y_to_x = min(y, capacity_x - x)
        possible_moves.add((x + pour_y_to_x, y - pour_y_to_x))
        for move in possible_moves:
            if move not in visited:
                queue.append(move)
                visited.add(move)
    print("No solution found!")
# Taking input from user
capacity_x = int(input("Enter capacity of first jug: "))
capacity_y = int(input("Enter capacity of second jug: "))
target = int(input("Enter target amount to measure: "))
bfs_water_jug(capacity_x, capacity_y, target)
```

Output:

```
Enter capacity of first jug: 15
Enter capacity of second jug: 12
Enter target amount to measure: 13
(0, 0)
(0, 12)
(15, 0)
(12, 0)
(15, 12)
(3, 12)
(12, 12)
(3, 0)
(15, 9)
(0, 3)
(0, 9)
(15, 3)
(9, 0)
(6, 12)
(9, 12)
(6, 0)
(15, 6)
(0, 6)
No solution found!
```

Experiment:8

Aim: Write a program to implement DFS using Python.

Language Used: Python

Theory:
Depth-First Search (DFS) is a graph traversal algorithm used to explore nodes and edges of a graph systematically. It starts at a given node and explores as far as possible along each branch before backtracking. DFS can be implemented using recursion or a stack. It is useful for solving problems like pathfinding, cycle detection, and topological sorting.

Source Code:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
      print(start, end=' ')
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
# Example graph representation using adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
# Taking input from user for starting node
start_node = input("Enter the starting node: ")
dfs(graph, start_node)
```

Output:

```
Enter the starting node: A
A B D E F C
```