MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

B.Tech FIFTH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
[B.Tech CSE(AI)]

ARTIFICIAL INTELLIGENCE LAB

[CSE_3182]

LABORATORY MANUAL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

Excellence in Computer Science and allied engineering disciplines through education, research, innovation and teamwork for emerging industry needs, societal aspirations, and sustainable global development.

Mission

Provide a conducive environment to acquire advanced skillset in computer science and allied engineering disciplines for professional roles in the emerging digital world, enhance knowledge through quality research, and establish a strong bond with industry, alumni, academia and society.

Program Outcomes:

[PO.1] Engineering Knowledge: Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

[PO.2] Problem Analysis: Identify, formulate, research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

[PO.3] Design/ Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.

[PO.4] Conduct investigations of complex problems using research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.

[PO.5] Modern Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modelling to complex engineering activities with an under- standing of the limitations.

[PO.6] The Engineer and Society: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to professional engineering practice.

[PO.7] Environment and Sustainability: Understand the impact of professional engineering solutions in societal and environmental contexts and demonstrate knowledge of and need for sustainable development.

[PO.8]. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

[PO.9] Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams and in multi-disciplinary settings.

[PO.10] Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
[PO.11] Project Management and Finance: Demonstrate knowledge and understanding of engineering and management principles and apply these to owners own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

[PO.12] Life-long Learning: Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

CONTENTS

Course Objectives

This laboratory aims to provide its learners the following as its objectives for hands on practical learning:

- To understand and apply of the concepts of informed and uninformed searching techniques.
- To interpret knowledge representation for its application in AI algorithms.
- To develop an Expert System and analyze it's working by interpretation using several facets of Artificial Intelligence Techniques ranging from conventional AI to Semantics Oriented AI.

Course Outcomes

At the end of this course, students will have the

- Ability to design one or more algorithms for a problem solving using appropriate Artificial Intelligence paradigms.
- Ability to formulate an algorithm into an efficient AI based techniques using production rules and conventional AI algorithms.
- Ability to hybridize AI techniques of different levels of complexity and build successful Expert Systems.
-

Evaluation Plan

- Internal Assessment Marks: 60 Marks

  Continuous evaluation: 30 Marks

  - The Continuous evaluation assessment will depend on punctuality, designing right program, converting algorithm into an efficient AI inclusive program, maintaining the class record and answering the questions in viva voce.
  - Mid Sem Exam : 15 Marks
  - Project Component on Building an Expert System : 15 Marks

- End semester assessment of 2 hour duration: 40 Marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
  - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
  - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
  - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
  - Statements within the program should be properly indented.
  - Use meaningful names for variables and functions.
  - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:

    - Lab exercises - to be completed during lab hours
    - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition lab with the permission of the faculty concerned.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.
- A sample note preparation is given as a model for observation.
- You may write scripts/programs to automate the experimental analysis of algorithms and compare with the theoretical result.
- You may use spreadsheets to plot the graph.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Artificial Intelligence Lab Exercises

Introduction:

The goal of this lab is to provide hands-on experience with different AI algorithms and techniques. You will implement and test different algorithms using Python and explore their strengths and limitations. The lab includes both programming and problem-solving exercises, as well as design and development of expert systems.

Requirements:

- A computer with Python 3 installed
- Jupyter Notebook or another Python development environment
- The Python libraries NumPy, Pandas, Matplotlib, Scikit-learn, and any other libraries required for specific exercises

1. Implementation of Depth First Search

a. Implement the DFS algorithm using Python.

Procedure:

Step 1: Create a stack and push the starting node into the stack.

Step 2: While the stack is not empty:

a. Pop the top element from the stack.

b. If the popped element is the goal node, return success.

c. If the popped element is not visited, mark it as visited.

d. Push all the neighbours of the popped element into the stack if they are not visited.

Program:

```
def dfs(node, graph, visited, component):
    component.append(node) # Store answer
    visited[node] = True  # Mark visited

    # Traverse to each adjacent node of a node
    for child in graph[node]:
        if not visited[child]: # Check whether the node is visited or not
            dfs(child, graph, visited, component)  # Call the dfs recursively

if__name__== "_main__":

    # Graph of nodes
    graph = {
        0: [2],
        1: [2, 3],
        2: [0, 1, 4],
        3: [1, 4],
        4: [2, 3]
    }
    node = 0  # Starting node
    visited = [False]*len(graph) # Make all nodes to False initially
    component = []
    dfs(node, graph, visited, component) # Traverse to each node of a graph
    print(f"Following is the Depth-first search: {component}")  # Print the answer
```

```
Following is the Depth-first search: [0, 2, 1, 3, 4]
```

Output:

2. Implementation of Breadth First Search

a. Implement the BFS algorithm using Python.

Procedure:

Step 1: Create an empty queue and enqueue the starting vertex.
Step 2: Create a set to keep track of visited vertices.

Step 3: While the queue is not empty:
a.Dequeue a vertex from the queue.

b.If the vertex has not been visited:

i.. Mark the vertex as visited.

ii. Add the vertex to the list of visited vertices.

iii. Enqueue all adjacent vertices of the dequeued vertex that have not been visited.

Step 4: Return the list of visited vertices.

Program

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue
```

```
def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
      m = queue.pop(0)
      print (m, end = " ")

      for neighbour in graph[m]:
          if neighbour not in visited:
              visited.append(neighbour)
              queue.append(neighbour)
# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

Output

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

3. Implementation of Uniform cost search

a. Implement the Uniform cost search algorithm using Python.

Procedure:

Step 1: Initialize a priority queue and insert the starting node with cost 0.

Step 2: Create an empty set to keep track of visited nodes.

Step 3: While the priority queue is not empty:

a. Remove the node with the lowest cost from the priority queue.

b. If the node is the goal node, return the solution path.

c. Add the node to the visited set. d. Expand the node and add its neighbors to the priority queue with the accumulated cost.

Step 4: If the priority queue becomes empty and the goal node is not reached, return failure.

Program:

```
from queue import PriorityQueue

def uniform_cost_search(start_node, goal_node, graph):

    frontier = PriorityQueue()

    frontier.put((0, start_node))

    explored = set()

    while not frontier.empty():
```

```
        current_cost, current_node = frontier.get()
        if current_node == goal_node:
            return current_cost
        explored.add(current_node)
        for neighbor, cost in graph[current_node].items():
            if neighbor not in explored:
                new_cost = current_cost + cost
                frontier.put((new_cost, neighbor))
    return -1 # Failure
```

b. Test your implementation on a simple graph and on a larger graph with multiple solutions.

Program:

```
# Simple graph
graph = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 2},
    'C': {'D': 4},
    'D': {}
}
start_node = 'A'
goal_node = 'D'
assert uniform_cost_search(start_node, goal_node, graph) == 7
# Larger graph with multiple solutions
graph = {
    'A': {'B': 2, 'C': 1},
    'B': {'D': 2},
    'C': {'D': 3, 'E': 4},
    'D': {'F': 1},
    'E': {'F': 5},
    'F': {}
}
start_node = 'A'
goal_node = 'F'
```

```
assert uniform_cost_search(start_node, goal_node, graph) == 5
# Simple graph
graph = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 2},
    'C': {'D': 4},
    'D': {}
}
start_node = 'A'
goal_node = 'D'
assert uniform_cost_search(start_node, goal_node, graph) == 7
# Larger graph with multiple solutions
graph = {
    'A': {'B': 2, 'C': 1},
    'B': {'D': 2},
    'C': {'D': 3, 'E': 4},
    'D': {'F': 1},
    'E': {'F': 5},
    'F': {}
}
start_node = 'A'
goal_node = 'F'
assert uniform_cost_search(start_node, goal_node, graph) == 5
```

Output:

```
7
5
```

4. Implementation of Hill climbing search

a. Implement the Hill climbing search algorithm using Python.

Procedure:

Step 1: Define the optimization problem to be solved.

Step 2: Initialize the current state as a random or pre-defined starting point.

Step 3: Evaluate the objective function for the current state.

Step 4: Generate a set of possible next states by making small modifications to the current state.

Step 5: Evaluate the objective function for each of the possible next states.

Step 6: Select the next state with the highest objective function value as the new current state.

Step 7: Repeat steps 4-6 until a stopping criterion is met or a maximum number of iterations is reached.

Step 8: Return the final state as the solution to the optimization problem.

Program:

```
import random
def hill_climbing_search(f, neighbor_fn, max_iter=1000):
  current = random.choice(list(x_range))
  for i in range(max_iter):
          neighbors = neighbor_fn(current)
          next_neighbor = max(neighbors, key=lambda x: f(x))
          if f(next_neighbor) <= f(current):
                  break
          current = next_neighbor
  return current, f(current)
def f(x):
  return -x**2
def neighbor_fn(x):
  return [x + dx for dx in [-0.1, 0, 0.1]]
x_range= [x for x in range(10)] # X range from 0 to 10
best_solution, best_value = hill_climbing_search(f, neighbor_fn)
print("Best solution: x =", best_solution, "Best value: f(x) =", -best_value)
```

b. Test your implementation on a simple optimization problem, such as finding the maximum of a function.

Program:

Example of using hill climbing search to find the maximum value of the function f(x) = x^2 in the range [0, 10]:

```
def f(x):
    return -x**2
def neighbor_fn(x):
    return [x + dx for dx in [-0.1, 0, 0.1]]
x_range= [x for x in range(10)] # X range from 0 to 10
best_solution, best_value = hill_climbing_search(f, neighbor_fn)
print("Best solution: x =", best_solution, "Best value: f(x) =", -best_value)
```

output:

```
Best solution: x = 0 Best value: f(x) = 0
```

5. Implementation of A* Algorithm

a. Implement the A* algorithm using Python. Procedure:

Step 1: Define the start node and the goal node

Step 2: Initialize the open list with the start node.

Step 3: Initialize the closed list as empty.

Step 4: While the open list is not empty:

a. Sort the open list by the total cost of each node, where the total cost is the sum of the cost from the start node to the current node and the heuristic cost from the current node to the goal node.

b. Get the node with the lowest total cost and add it to the closed list.

c. If the current node is the goal node, return the path.

d. For each neighbor of the current node:

i. If the neighbor is not already in the closed list, calculate its total cost and add it to the open list.

ii. If the neighbor is already in the open list, update its total cost if the new total cost is lower than the previous one.

Program:

```
import heapq

def astar(graph, start, goal, heuristic):

    # Initialize the frontier and explored set

    frontier = [(heuristic[start], start)]

    explored = set()

    # Initialize the cost and path dictionaries

    cost = {start: 0}

    path = {start: None}


    while frontier:

        # Pop the node with the Lowest cost from the frontier

        _, current = heapq.heappop(frontier)

        if current == goal:
```

```
                # Build the path from start to goal
                path_list = [current]
                while path[current] != None:
                    path_list.append(path[current])
                    current = path[current]
                path_list.reverse()
                return path_list

            explored. add(current)
            # Expand the neighbors of the current node
            for neighbor in graph[current]:
                new_cost = cost[current] + graph[current] [neighbor]
                if neighbor not in cost or new_cost < cost[neighbor]:
                    cost[neighbor] = new_cost
                    priority = new_cost + heuristic[neighbor]
                    heapq. heappush (frontier, (priority, neighbor))
                    path[neighbor] = current
        # If the goal is not reached, return None
        return None
# Define the graph as a dictionary
graph = {
'A' : {'B':5, 'C':10},
'B' : {'D':15},
'C' : {'D':20},'D':{}
}
#Define the Heuristic as Dictionary
heuristic = {
'A':15,
'B':10,
'C':5,
```

```
'D':0

}


# Test the A* algorithm

start = 'A'

goal = 'D'

path = astar(graph, start, goal, heuristic)

print("shortest path from ", start, " to ", goal, " : ", path)
```

Output:

```
shortest path from  A  to  D  :  ['A', 'B', 'D']
```

6. Implementation of Crypt Arithmetic

Implement a program to solve a cryptarithmetic puzzle using Python.

Procedure:

Step 1: Define the problem by identifying the letters that represent digits, and the arithmetic operations that need to be performed.

Step 2: Generate all possible assignments of digits to the letters.

Step 3: For each assignment, check if it satisfies the problem constraints (i.e., performs the correct arithmetic operations).

Step 4: If an assignment satisfies the constraints, return the solution.

Step 5: If no assignment satisfies the constraints, return "No solution found".

Program:

```
import itertools

def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor * substitution[letter]
        factor *= 10
    return s

def solve(equation):
    # split equation in left and right
    left, right = equation.lower().replace(' ', '').split('=')
    # split words in left part
    left = left.split('+')
    # create list of used letters
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)

    digits = range(10)
    for perm in itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))

        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
            print(' + '.join(str(get_value(word, sol)) for word in left) + " = {} (mapping:
{})".format(get_value(right, sol), sol))
```

if__name__== '_main_':
  solve('SEND + MORE = MONEY')

a. Test your program on different puzzles

SEND

+ MORE

MONEY

Program:

solution = solve('SEND + MORE = MONEY')

print(solution)

Output:

```
if __name__ == '__main__':
    solve('SEND + MORE = MONEY')

8324 + 913 = 9237 (mapping: {'e': 3, 'm': 0, 'd': 4, 'n': 2, 's': 8, 'r': 1, 'o': 9, 'y': 7})
7316 + 823 = 8139 (mapping: {'e': 3, 'm': 0, 'd': 6, 'n': 1, 's': 7, 'r': 2, 'o': 8, 'y': 9})
8432 + 914 = 9346 (mapping: {'e': 4, 'm': 0, 'd': 2, 'n': 3, 's': 8, 'r': 1, 'o': 9, 'y': 6})
6415 + 734 = 7149 (mapping: {'e': 4, 'm': 0, 'd': 5, 'n': 1, 's': 6, 'r': 3, 'o': 7, 'y': 9})
6419 + 724 = 7143 (mapping: {'e': 4, 'm': 0, 'd': 9, 'n': 1, 's': 6, 'r': 2, 'o': 7, 'y': 3})
7429 + 814 = 8243 (mapping: {'e': 4, 'm': 0, 'd': 9, 'n': 2, 's': 7, 'r': 1, 'o': 8, 'y': 3})
7531 + 825 = 8356 (mapping: {'e': 5, 'm': 0, 'd': 1, 'n': 3, 's': 7, 'r': 2, 'o': 8, 'y': 6})
8542 + 915 = 9457 (mapping: {'e': 5, 'm': 0, 'd': 2, 'n': 4, 's': 8, 'r': 1, 'o': 9, 'y': 7})
6524 + 735 = 7259 (mapping: {'e': 5, 'm': 0, 'd': 4, 'n': 2, 's': 6, 'r': 3, 'o': 7, 'y': 9})
7534 + 825 = 8359 (mapping: {'e': 5, 'm': 0, 'd': 4, 'n': 3, 's': 7, 'r': 2, 'o': 8, 'y': 9})
7539 + 815 = 8354 (mapping: {'e': 5, 'm': 0, 'd': 9, 'n': 3, 's': 7, 'r': 1, 'o': 8, 'y': 4})
9567 + 1085 = 10652 (mapping: {'e': 5, 'm': 1, 'd': 7, 'n': 6, 's': 9, 'r': 8, 'o': 0, 'y': 2})
7643 + 826 = 8469 (mapping: {'e': 6, 'm': 0, 'd': 3, 'n': 4, 's': 7, 'r': 2, 'o': 8, 'y': 9})
7649 + 816 = 8465 (mapping: {'e': 6, 'm': 0, 'd': 9, 'n': 4, 's': 7, 'r': 1, 'o': 8, 'y': 5})
5731 + 647 = 6378 (mapping: {'e': 7, 'm': 0, 'd': 1, 'n': 3, 's': 5, 'r': 4, 'o': 6, 'y': 8})
3712 + 467 = 4179 (mapping: {'e': 7, 'm': 0, 'd': 2, 'n': 1, 's': 3, 'r': 6, 'o': 4, 'y': 9})
5732 + 647 = 6379 (mapping: {'e': 7, 'm': 0, 'd': 2, 'n': 3, 's': 5, 'r': 4, 'o': 6, 'y': 9})
3719 + 457 = 4176 (mapping: {'e': 7, 'm': 0, 'd': 9, 'n': 1, 's': 3, 'r': 5, 'o': 4, 'y': 6})
3821 + 468 = 4289 (mapping: {'e': 8, 'm': 0, 'd': 1, 'n': 2, 's': 3, 'r': 6, 'o': 4, 'y': 9})
6851 + 738 = 7589 (mapping: {'e': 8, 'm': 0, 'd': 1, 'n': 5, 's': 6, 'r': 3, 'o': 7, 'y': 9})
6853 + 728 = 7581 (mapping: {'e': 8, 'm': 0, 'd': 3, 'n': 5, 's': 6, 'r': 2, 'o': 7, 'y': 1})
2817 + 368 = 3185 (mapping: {'e': 8, 'm': 0, 'd': 7, 'n': 1, 's': 2, 'r': 6, 'o': 3, 'y': 5})
2819 + 368 = 3187 (mapping: {'e': 8, 'm': 0, 'd': 9, 'n': 1, 's': 2, 'r': 6, 'o': 3, 'y': 7})
3829 + 458 = 4287 (mapping: {'e': 8, 'm': 0, 'd': 9, 'n': 2, 's': 3, 'r': 5, 'o': 4, 'y': 7})
5849 + 638 = 6487 (mapping: {'e': 8, 'm': 0, 'd': 9, 'n': 4, 's': 5, 'r': 3, 'o': 6, 'y': 7})
```

7. Implementation of Water jug problem

Procedure:

Step 1: Start with the initial state of two empty jugs, Jug1 and Jug2, and the goal state of a specific amount of water in one of the jugs.

Step 2: Apply the search algorithm to find a path from the initial state to the goal state, using the following operations:

a. Fill a jug to its maximum capacity.

b. Empty a jug completely.

c. Pour water from one jug to another until the receiving jug is full or the source jug is empty.

Step 3: Once the goal state is reached, the search algorithm can be terminated, and the path from the initial state to the goal state can be returned.

Program:

```
def solve(jug_sizes, target_volume, start_volumes):
    explored = set()
    start_state = tuple(start_volumes)
    path = dfs(start_state, jug_sizes, target_volume, explored)
    return path

def dfs(current_state, jug_sizes, target_volume, explored):
    jug1, jug2 = current_state

    if jug1 == target_volume or jug2 == target_volume:
        return [current_state]

    explored.add(current_state)

    successors = get_successors(current_state, jug_sizes)

    for successor in successors:
        if successor not in explored:
            path = dfs(successor, jug_sizes, target_volume, explored)
            if path is not None:
                return [current_state] + path

    return None

def get_successors(state, jug_sizes):
    jug1, jug2 = state
    jug1_cap, jug2_cap = jug_sizes

    successors = []

    successors.append((jug1_cap, jug2))
    successors.append((jug1, jug2_cap))
```

```
        successors.append((0, jug2))
        successors.append((jug1, 0))

        amount_to_pour = min(jug1, jug2_cap - jug2)
        successors.append((jug1 - amount_to_pour, jug2 + amount_to_pour))

        amount_to_pour = min(jug2, jug1_cap - jug1)
        successors.append((jug1 + amount_to_pour, jug2 - amount_to_pour))

        return [s for s in successors if is_valid_state(s, jug_sizes)]

    def is_valid_state(state, jug_sizes):
        jug1_cap, jug2_cap = jug_sizes
        jug1, jug2 = state
        return 0 <= jug1 <= jug1_cap and 0 <= jug2 <= jug2_cap

    jugs = (4, 3)
    target = 2
    start = (0, 0)
    path = solve(jugs, target, start)
    print("Test case 1:")
    print(path)  # Expected output: [(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2)]
```

b. Test your program on different jug sizes and target volumes.

To test the water jug problem implementation, we can create different scenarios with different jug sizes and target volumes. Here are a few examples:

Scenario 1:

Jug sizes: 4 gallons, 3 gallons

Target volume: 2 gallons

Expected solution: (4, 2)

Scenario 2:

Jug sizes: 4 gallons, 6 gallons

Target volume: 5 gallons

Expected solution: None

Scenario 3:

Jug sizes: 2 gallons, 7 gallons

Target volume: 3 gallons

Expected solution: No solution exists

Scenario 4:

Jug sizes: 5 gallons, 9 gallons

Target volume: 3 gallons

Expected solution: (2, 3)

Program:

```
# Test Water Jug Problem implementation from water_jug_problem import solve

# Test cases:

jugs = (4, 3)
target = 2
start = (0, 0)
path = solve(jugs, target, start)
print("Test case 1:")
print(path)  # Expected output: [(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2)]

jugs = (4, 6)
target = 5
start = (0, 0)
path = solve(jugs, target, start)
print("Test case 2:")
print(path)  # Expected output: None

jugs = (2, 7)
target = 3
start = (0, 0)
path = solve(jugs, target, start)
print("Test case 3:")
print(path)  # Expected output: [(0, 0), (2, 0), (2, 7), (0, 7), (2, 5), (0, 5), (2, 3)]

jugs = (5, 9)
target = 3
start = (0, 0)
path = solve(jugs, target, start)
print("Test case 4:")
print(path) # Expected output: [(0, 0), (5, 0), (5, 9), (0, 9), (5, 4), (0, 4), (4, 0), (4, 9), (5, 8), (0, 8), (5, 3)]
```

Output:

```
Test case 1:
[(0, 0), (4, 0), (4, 3), (0, 3), (3, 0), (3, 3), (4, 2)]
Test case 2:
None
Test case 3:
[(0, 0), (2, 0), (2, 7), (0, 7), (2, 5), (0, 5), (2, 3)]
Test case 4:
[(0, 0), (5, 0), (5, 9), (0, 9), (5, 4), (0, 4), (4, 0), (4, 9), (5, 8), (0, 8), (5, 3)]
```

8. Implementation of Missionaries and Cannibals problem

a. Implement a program to solve the missionaries and cannibals problem using Python.

Step 1: Create an initial state with the current number of missionaries and cannibals on each side of the river.

Step 2: Create a goal state where all the missionaries and cannibals are on the opposite side of the river.

Step 3: Create a list of valid actions that can be taken from the current state. Valid actions involve moving 1 or 2 people from one side of the river to the other.

Step 4: Apply each valid action to the current state to create a list of possible successor states.

Step 5: Check if each successor state is valid. A state is valid if there are no more cannibals than missionaries on either side of the river.

Step 6: If a successor state is valid, add it to a list of candidate states.

Step 7: Use a search algorithm, such as breadth-first search or depth-first search, to find a path from the initial state to the goal state.

Step 8: Return the path from the initial state to the goal state.

Program:

```
def valid(state):
    if state[0][0] < state[0][1] and state[0][0] > 0:
        return False
    if state[1][0] < state[1][1] and state[1][0] > 0:
        return False
    return True
def successors(state):
    children = []
    for i in range(3):
        for j in range(3):
            if i + j < 1 or i + j > 2:
                continue
            if state[2] == 1:
                child = ((state[0][0] - i, state[0][1] - j), (state[1][0] + i, state[1][1] + j), 0)
            else:
                child = ((state[0][0] + i, state[0][1] + j), (state[1][0] - i, state[1][1] - j), 1)
```

```
            if valid(child):
                children.append(child)
        return children
    def bfs(start, goal):
        visited = set() # Using a set for faster membership check
        queue = [[start]]
        while queue:
            path = queue.pop(0)
            node = path[-1]

            if node == goal:
                return path
            for child in successors(node):
                if child not in visited:
                    visited.add(child)
                    new_path = list(path)
                    new_path.append(child)
                    queue.append(new_path)
        return []
    initial = ((3, 3), (0, 0), 1)
    goal = ((0, 0), (3, 3), 0)
    path = bfs(initial, goal)

    if path:
        for state in path:
            print(state)
    else:
        print("No solution found.")
```

b. Test your program on different initial and goal states.

Program:

```
# Test case 1
initial = ((3,3), (0,0), 1)
```

goal = ((0,0), (3,3), 0)

Test case 2

initial = ((3,2), (0,1), 1)

goal = ((0,1), (3,2), 0)

Test case 3

initial = ((3,1), (0,2), 1)

goal = ((0,2), (3,1), 0)

Output for the 3 test cases:

```
((3, 3), (0, 0), 1)
((3, 1), (0, 2), 0)
((3, 2), (0, 1), 1)
((3, 0), (0, 3), 0)
((3, 1), (0, 2), 1)
((1, 1), (2, 2), 0)
((2, 2), (1, 1), 1)
((0, 2), (3, 1), 0)
((0, 3), (3, 0), 1)
((0, 1), (3, 2), 0)
((0, 2), (3, 1), 1)
((0, 0), (3, 3), 0)


((3, 2), (0, 1), 1)
((3, 0), (0, 3), 0)
((3, 1), (0, 2), 1)
((1, 1), (2, 2), 0)
((2, 2), (1, 1), 1)
((0, 2), (3, 1), 0)
((0, 3), (3, 0), 1)
((0, 1), (3, 2), 0)


((3, 1), (0, 2), 1)
((1, 1), (2, 2), 0)
((2, 2), (1, 1), 1)
((0, 2), (3, 1), 0)
```

9. Implementation of 8 queen's problem

Procedure:

Step 1: Start in the leftmost column

Step 2: If all queens are placed, return true

Step 3: Try all rows in the current column. For each row, a. If the queen can be placed safely in this row, mark this [row,column] as part of the solution and recursively check if placing queen here leads to a solution. b. If placing the queen in [row, column] leads to a solution, return true. c. If placing the queen in [row, column] does not lead to a solution, unmark this [row, column] and go to step 3(b) to try the next row.

Step 4: If all rows have been tried and nothing worked, return false to trigger backtracking.

Program:

```
def is_safe(board, row, col, n):

    # Check if there is any queen in the same row

    for i in range(col):

        if board[row][i] == 1:

            return False

    # Check upper diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    # Check lower diagonal

    for i, j in zip(range(row, n), range(col, -1, -1)):

        if board[i][j] == 1:

            return False
```

```
    # If there is no queen in the same row or in the diagonals, then it is safe to place a queen
    return True
def solve_n_queens(board, col, n):
    # Base case: If all queens are placed successfully
    if col == n:
        print_board(board)
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            # Recursively call solve_n_queens function for the next column
            if solve_n_queens(board, col + 1, n):
                return True
            # If solve_n_queens function does not return True, remove the queen from the current cell
            board[i][col] = 0
    # If a queen cannot be placed in any row in the current column, return False
    return False
def print_board(board):
    for row in board:
        print(row)
# Testing the program for different board sizes
n = 4
board = [[0 for _ in range(n)] for _ in range(n)]
```

solve_n_queens(board, 0, n)

n = 8

board = [[0 for _ in range(n)] for _ in range(n)]

solve_n_queens(board, 0, n)

Output:

```
      [0, 0, 1, 0]
      [1, 0, 0, 0]
      [0, 0, 0, 1]
      [0, 1, 0, 0]
[5]:  True
      [1, 0, 0, 0, 0, 0, 0, 0]
      [0, 0, 0, 0, 0, 0, 1, 0]
      [0, 0, 0, 0, 1, 0, 0, 0]
      [0, 0, 0, 0, 0, 0, 0, 1]
      [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, 1, 0, 0, 0, 0, 0]
[4]:  True
```

10. Implementation of Best First Search

a. Implement the Best First Search algorithm using Python.

Step 1: Create a queue to store nodes to be expanded and a set to store visited nodes.

Step 2: Add the starting node to the queue and mark it as visited.

Step 3: While the queue is not empty:

a. Pop the node with the lowest heuristic value from the queue.

b. If the node is the goal node, return the path to it.

c. Otherwise, expand the node and add its unvisited children to the queue, marking them as visited and assigning them heuristic values.

Step 4: If the queue is empty and the goal node has not been found, return failure.

Program:

```
import heapq


def best_first_search(start, goal, heuristics, graph):
    open_list = [(heuristics[start], start, [start])]
    visited = set()
    while open_list:
        _, current_node, current_path = heapq.heappop(open_list)
        if current_node == goal:
            return current_path # Path found
        visited.add(current_node)
        for neighbor, cost in graph.get(current_node, []):
            if neighbor not in visited:
                heapq.heappush(open_list, (heuristics[neighbor], neighbor, current_path + [neighbor]))
    return None # No path found
# Example heuristics dictionary and graph dictionary
heuristics = {
    'A': 5,
    'B': 4,
    'C': 3,
    'D': 2,
    'E': 0,
}
```

```
graph = {
    'A': [('B', 3), ('C', 2)],
    'B': [('D', 4)],
    'C': [('D', 1)],
    'D': [('E', 3)],
    'E': []
}

start_node = 'A'
goal_node = 'E'

path = best_first_search(start_node, goal_node, heuristics, graph)
if path:
    print("Path found:", " -> ".join(path))
else:
    print("Path not found.")
```

b. Test your implementation on a simple graph and on a larger graph with multiple solutions.

Program

```
# Simple graph example
graph = {
    'A': [('B', 3), ('C', 2)],
    'B': [('D', 4)],
    'C': [('D', 1)],
    'D': [('E', 3)],
    'E': []
}
# Heuristic function for simple graph
heuristic = {
    'A': 5,
    'B': 4,
    'C': 3,
```

```
    'D': 2,
    'E': 0
}
# Start and goal nodes for simple graph
start_node = 'A'
goal_node = 'E'


# Testing the Best First Search algorithm on simple graph
# Larger graph example
graph = {
    'S': [('A', 3), ('B', 6), ('C', 2)],
    'A': [('D', 3), ('E', 2)],
    'B': [('E', 1), ('F', 2)],
    'C': [('G', 6)],
    'D': [('H', 4), ('I', 3)],
    'E': [('J', 4)],
    'F': [('J', 2)],
    'G': [('J', 5)],
    'H': [('K', 2)],
    'I': [('L', 4)],
    'J': [('M', 3)],
    'K': [('N', 3)],
    'L': [('N', 2)],
    'M': [('N', 5)],
    'N': []
}
# Heuristic function for larger graph
heuristic = {
    'S': 14,
    'A': 11,
    'B': 10,
    'C': 6,
```

```
    'D': 7,
    'E': 4,
    'F': 4,
    'G': 5,
    'H': 2,
    'I': 4,
    'J': 3,
    'K': 1,
    'L': 2,
    'M': 0,
    'N': 0
}
# Start and goal nodes for larger graph
start_node = 'S'
goal_node = 'N'
```

Output:

Path found: ['A', 'C', 'D', 'E']

Path found: ['S', 'C', 'G', 'J', 'M', 'N']

```
Path found: A -> C -> D -> E
```

```
Path found: S -> C -> G -> J -> M -> N
```

11 and 12 Design and Development of expert system for the specified domain

Step 1: Choose a domain for your expert system, such as medical diagnosis, financial planning, or travel recommendations.

Step 2: Define the knowledge representation and reasoning methods for your expert system.

Step 3: Implement the expert system using Python and test it on different scenarios.

Step 4: Compare and Justify the features of the system developed using AI Principles from ranging from Statistical AI to Semantic AI.

Conclusion:

This lab provided an overview of different AI algorithms and techniques and allowed you to gain practical experience with their implementation and application. You have learned about various search algorithms such as Depth First Search, Breadth First Search, Uniform Cost Search, Hill Climbing Search, and A* Algorithm. Additionally, you have implemented some classical AI problems such as Crypt Arithmetic, Water Jug Problem, Missionaries and Cannibals Problem, and 8 Queens Problem. Moreover, you have gained knowledge about expert systems and how they can be designed and developed for different domains. Overall, this lab has helped you develop skills in problem-solving, critical thinking, and algorithmic design, which are essential for any AI practitioner. These skills can be further refined by practicing and implementing more complex AI problems and algorith