AI Lab Manual

Artificial Intelligence (East Point College of Engineering and Technology)

EAST POINT COLLEGE OF ENGINEERING & TECHNOLOGY

Department of Artificial Intelligence and Data Science

'Jnana Prabha', Virgo Nagar Post, Bengaluru-560049

Academic Year: 2023-24

LABORATORY MANUAL

SEMESTER : IV

SUBJECT : Artificial Intelligence

SUBCODE : BAD402

NAME: ____________________

USN: ____________________

SECTION: ____________________

BATCH: ____________________

@#

INSTITUTE VISION AND MISSION

VISION

The East Point College of Engineering and Technology aspires to be a globally acclaimed institution, recognized for excellence in engineering education, applied research and nurturing students for holistic development.

MISSION

M1: To create engineering graduates through quality education and to nurture innovation, creativity and excellence in teaching, learning and research

M2: To serve the technical, scientific, economic and societal developmental needs of our communities

M3: To induce integrity, teamwork, critical thinking, personality development and ethics in students and to lay the foundation for lifelong learning

Department of Artificial Intelligence and Data Science

DEPARTMENT VISION AND MISSION

VISION

The department orients towards identifying and exploring emerging global trends in the fields of Artificial Intelligence and Data Science through academic excellence and quality research, producing proficient professionals for a flourishing society.

MISSION

M1: To nurture students with quality education, life-long learning, values and ethics.

M2: To produce ethical and competent professionals through comprehensive and holistic methodologies that align with the global industry demands in Artificial Intelligence and Data Science.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. Graduates will possess the ability to apply their knowledge of fundamental engineering, Computer Science and Data Science.
2. Graduates will have sound intercommunication skills, ethical values and responsibilities to work and serve for the development of the society.
3. Graduates will be able to understand, interpret, model and implement the Artificial Intelligence and Data Science based solutions for real world problems.

PROGRAM SPECIFIC OUTCOMES (PSOs)

1. To cater and enhance the analytical and technical skills of the graduates in order to be ready for the professional development, research and pursue higher education.
2. To formulate solutions for the real-world problems with the application of basic engineering principles and technical skills of Artificial Intelligence and Data Science.

ARTIFICIAL INTELLIGENCE		Semester	IV
Course Code	BAD402	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100

@#

Credits	04	Exam Hours	
Examination nature (SEE)	Theory/		

Course objectives:

- Gain a historical perspective of AI and its foundations.
- Become familiar with basic principles of AI toward problem solving
- Get to know approaches of inference, perception, knowledge representation, and learning

PRACTICAL COMPONENT OF IPCC*(May cover all / major modules)*

NOTE: Programs need to be implemented in python

Sl.NO	Experiments
1	Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem
2	Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python
3	Implement A* Search algorithm
4	Implement AO* Search algorithm
5	Solve 8-Queens Problem with suitable assumptions
6	Implementation of TSP using heuristic approach
7	Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining
8	Implement resolution principle on FOPL related problems
9	Implement Tic-Tac-Toe game using Python
10	Build a bot which provides all the information related to text in search box
11	Implement any Game and demonstrate the Game playing strategies

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

CO1: Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO 2. Compare various Searching and Inferencing Techniques.
CO 3. Develop knowledge base sentences using propositional logic and first order logic
CO 4. Describe the concepts of quantifying uncertainty.
CO5: Use the concepts of Expert Systems to build applications.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks)**.
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

- **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
- The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.

- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scoredby the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Text Books

1. Stuart J. Russell and Peter Norvig , Artificial Intelligence, 3rd Edition, Pearson,2015
2. Elaine Rich, Kevin Knight, Artificial Intelligence, 3rd edition,Tata McGraw Hill,2013

Reference:

1. George F Lugar, Artificial Intelligence Structure and strategies for complex, Pearson Education, 5th Edition, 2011
2. Nils J. Nilsson, Principles of Artificial Intelligence, Elsevier, 1980
3. Saroj Kaushik, Artificial Intelligence, Cengage learning, 2014

Web links and Video Lectures (e-Resources)

1. https://www.kdnuggets.com/2019/11/10-free-must-read-books-ai.html
2. https://www.udacity.com/course/knowledge-based-ai-cognitive-systems--ud409
3. https://nptel.ac.in/courses/106/105/106105077/

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

1. Group discussion on Real world examples
2. Project based learning
3. Simple strategies on gaming, reasoning and uncertainty etc

@#

1.Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

```
class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2
    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2
    def __hash__(self):
        return hash((self.jug1, self.jug2))
   def __str__(self):
        return f"({self.jug1},{self.jug2})"
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
  def path(self):
        if self.parent is None:
            return [self.state]
        else:
            return self.parent.path() + [self.state]
def dfs(start_state, goal):
    visited = set()
    stack = [Node(start_state)]
  while stack:
        node = stack.pop()
        state = node.state
        if state == goal:
            return node.path()
  visited.add(state)
```

```
    # Actions: Fill jug1, fill jug2, empty jug1, empty jug2, pour from jug1 to jug2, pour from jug2 to jug1
    actions = [(state.jug1, 4), (4, state.jug2), (0, state.jug2), (state.jug1, 0),
               (min(state.jug1 + state.jug2, 4), max(0, state.jug1 + state.jug2 - 4)),
               (max(0, state.jug1 + state.jug2 - 3), min(state.jug1 + state.jug2, 3))]
   for action in actions:
      new_state = State(action[0], action[1])
      if new_state not in visited:
         stack.append(Node(new_state, node))
  return None
# Test the algorithm with an example
start_state = State(0, 0)  # Initial state: both jugs are empty
goal_state = State(2, 0)   # Goal state: jug1 has 2 units of water

print("Starting DFS for Water Jug Problem...")
path = dfs(start_state, goal_state)

if path:
   print("Solution found! Steps to reach the goal:")
   for i, state in enumerate(path):
      print(f"Step {i}: Jug1: {state.jug1}, Jug2: {state.jug2}")
else:
  print("No solution found!")
```

Output:

Starting DFS for Water Jug Problem...

Solution found! Steps to reach the goal:

Step 0: Jug1: 0, Jug2: 0

Step 1: Jug1: 4, Jug2: 0

Step 2: Jug1: 1, Jug2: 3

Step 3: Jug1: 1, Jug2: 0

Step 4: Jug1: 0, Jug2: 1

Step 5: Jug1: 4, Jug2: 1

Step 6: Jug1: 2, Jug2: 3

Step 7: Jug1: 2, Jug2: 0

2) Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems

```
from queue import PriorityQueue

# State representation: (left_missionaries, left_cannibals, boat_position)

INITIAL_STATE = (3, 3, 1)

GOAL_STATE = (0, 0, 0)

def is_valid_state(state):

    left_missionaries, left_cannibals, boat_position = state

    right_missionaries = 3 - left_missionaries

    right_cannibals = 3 - left_cannibals

   # Check if missionaries are outnumbered by cannibals on either side

    if left_missionaries > 0 and left_cannibals > left_missionaries:

        return False

    if right_missionaries > 0 and right_cannibals > right_missionaries:

        return False

        return True

 def generate_next_states(state):

    next_states = []

    left_missionaries, left_cannibals, boat_position = state

    new_boat_position = 1 - boat_position

        for m in range(3):

        for c in range(3):
```

```
            if 1 <= m + c <= 2:
                if boat_position == 1:
                    new_left_m = left_missionaries - m
                    new_left_c = left_cannibals - c
                else:
                    new_left_m = left_missionaries + m
                    new_left_c = left_cannibals + c
                new_state = (new_left_m, new_left_c, new_boat_position)
                if is_valid_state(new_state):
                    next_states.append(new_state)
        return next_states
def bfs():
    frontier = PriorityQueue()
    frontier.put((0, INITIAL_STATE))
    came_from = {}
    cost_so_far = {INITIAL_STATE: 0}
  while not frontier.empty():
        _, current_state = frontier.get()
    if current_state == GOAL_STATE:
            break
      for next_state in generate_next_states(current_state):
            new_cost = cost_so_far[current_state] + 1
            if next_state not in cost_so_far or new_cost < cost_so_far[next_state]:
```

```
                cost_so_far[next_state] = new_cost

                priority = new_cost

                frontier.put((priority, next_state))

                came_from[next_state] = current_state


    # Reconstruct path

    current_state = GOAL_STATE

    path = [current_state]

    while current_state != INITIAL_STATE:

        current_state = came_from[current_state]

        path.append(current_state)

    path.reverse()

  return path

def print_path(path):

    for i, state in enumerate(path):

        left_missionaries, left_cannibals, boat_position = state

        right_missionaries = 3 - left_missionaries

        right_cannibals = 3 - left_cannibals

        print(f"Step {i}: ({left_missionaries}M, {left_cannibals}C, {'left' if boat_position == 1
  else 'right'}) "

              f"-> ({right_missionaries}M, {right_cannibals}C, {'right' if boat_position == 1 else
  'left'})")

  if __name__ == "__main__":
```

path = bfs()

print("Solution path:")

print_path(path)

output

```
Solution path:
Step 0: (3M, 3C, left) -> (0M, 0C, right)
Step 1: (2M, 2C, right) -> (1M, 1C, left)
Step 2: (3M, 2C, left) -> (0M, 1C, right)
Step 3: (3M, 0C, right) -> (0M, 3C, left)
Step 4: (3M, 1C, left) -> (0M, 2C, right)
Step 5: (1M, 1C, right) -> (2M, 2C, left)
Step 6: (2M, 2C, left) -> (1M, 1C, right)
Step 7: (0M, 2C, right) -> (3M, 1C, left)
Step 8: (0M, 3C, left) -> (3M, 0C, right)
Step 9: (-1M, 2C, right) -> (4M, 1C, left)
Step 10: (0M, 2C, left) -> (3M, 1C, right)
Step 11: (0M, 0C, right) -> (3M, 3C, left)
```

3. Implement A* Search algorithm

```
import heapq
class Node:
    def __init__(self, state, parent=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def total_cost(self):
        return self.cost + self.heuristic

def astar_search(start_state, goal_state, neighbors_fn, heuristic_fn):
    open_set = []
    closed_set = set()

    start_node = Node(start_state, None, 0, heuristic_fn(start_state))
    heapq.heappush(open_set, (start_node.total_cost(), id(start_node), start_node))

    while open_set:
        _, _, current_node = heapq.heappop(open_set)

        if current_node.state == goal_state:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
```

```
            return path[::-1]
        closed_set.add(current_node.state)

        for neighbor_state in neighbors_fn(current_node.state):
            if neighbor_state in closed_set:
                continue

            neighbor_node = Node(neighbor_state)
            neighbor_node.parent = current_node
            neighbor_node.cost = current_node.cost + 1  # Assuming uniform cost
            neighbor_node.heuristic = heuristic_fn(neighbor_state)

            if any(neighbor_node.state == node.state for _, _, node in open_set):
                continue
           heapq.heappush(open_set,(neighbor_node.total_cost(),id(neighbor_node), neighbor_node)
           return None
# Example usage:
def neighbors(state):
    # Example: state is represented as a tuple (x, y)
    x, y = state
    # Define possible movements (up, down, left, right)
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    return [(x + dx, y + dy) for dx, dy in movements]
```

```
def heuristic(state):
    # Example: state is represented as a tuple (x, y)
    x, y = state
    # Calculate Manhattan distance to the goal state (0, 0)
    return abs(x) + abs(y)
start_state = (0, 0)
goal_state = (4, 4)
path = astar_search(start_state, goal_state, neighbors, heuristic)
print("Path:", path)
```

output

```
Path: [(0, 0), (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (4, 2), (4, 3),
(4, 4)]
```

4. Implement AO* Search algorithm

```
import heapq
class Node:
    def __init__(self, state, parent=None, cost=0, g=0, h=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.g = g
        self.h = h
    def total_cost(self):
        return self.g + self.h
  def ao_star_search(start_state, goal_state, neighbors_fn, heuristic_fn, epsilon):
    open_set = []
    closed_set = set()
  start_node = Node(start_state, None, 0, 0, heuristic_fn(start_state))
    heapq.heappush(open_set, (start_node.total_cost(), id(start_node), start_node))
    while open_set:
     _, _, current_node = heapq.heappop(open_set)
         if current_node.state == goal_state:
          path = []
          while current_node:
```

```
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]
    closed_set.add(current_node.state)
        for neighbor_state in neighbors_fn(current_node.state):
            if neighbor_state in closed_set:
                continue
            neighbor_node = Node(neighbor_state)
            neighbor_node.parent = current_node
            neighbor_node.g = current_node.g + 1  # Assuming uniform cost
            neighbor_node.h = heuristic_fn(neighbor_state)
            if any(neighbor_node.state == node.state for _, _, node in open_set):
                continue
            heapq.heappush(open_set, (neighbor_node.total_cost() + epsilon * neighbor_node.h,
id(neighbor_node), neighbor_node))
    return None
# Example usage:
def neighbors(state):
    # Example: state is represented as a tuple (x, y)
    x, y = state
    # Define possible movements (up, down, left, right)
    movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    return [(x + dx, y + dy) for dx, dy in movements]
```

```
def heuristic(state):
    # Example: state is represented as a tuple (x, y)
    x, y = state
    # Calculate Manhattan distance to the goal state (0, 0)
    return abs(x) + abs(y)
start_state = (0, 0)
goal_state = (4, 4)
epsilon = 1.0
path = ao_star_search(start_state, goal_state, neighbors, heuristic, epsilon)
print("Path:", path)
```

output

```
Path: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4)]
```

5. Solve 8-Queens Problem with suitable assumptions

```
def is_safe(board, row, col):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i] == col:
            return False
    # Check upper diagonal on left side
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i] == j:
            return False
     # Check upper diagonal on right side
    for i, j in zip(range(row-1, -1, -1), range(col+1, 8)):
        if board[i] == j:
            return False
      return True
    def solve_queens_util(board, row):
    if row >= 8:  # All queens are placed
        return True
      for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if solve_queens_util(board, row + 1):
```

```
            return True
        # If placing queen in board[i][col] doesn't lead to a solution, backtrack
        board[row] = -1
   return False
def solve_queens():
    board = [-1] * 8  # Initialize board with -1 indicating no queen placed in that row
    if not solve_queens_util(board, 0):
        print("Solution does not exist")
        return False
    print("Solution:")
    for i in range(8):
        for j in range(8):
            if board[i] == j:
                print("Q", end=" ")
            else:
                print(".", end=" ")
        print()
        return True
# Solve the 8-Queens Problem
solve_queens()
```

output

```
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

6. Implementation of TSP using heuristic approach

```
import numpy as np

def tsp_nearest_neighbor(distances):
    num_cities = distances.shape[0]
    visited = [False] * num_cities
    tour = []
    # Start from the first city
    current_city = 0
    tour.append(current_city)
    visited[current_city] = True
   # Visit each city
    for _ in range(num_cities - 1):
        nearest_city = None
        nearest_distance = float('inf')
   # Find the nearest unvisited city
        for next_city in range(num_cities):
            if not visited[next_city] and distances[current_city, next_city] < nearest_distance:
                nearest_city = next_city
                nearest_distance = distances[current_city, next_city]
    # Move to the nearest city
        current_city = nearest_city
        tour.append(current_city)
        visited[current_city] = True
   # Return to the starting city
    tour.append(tour[0])
    return tour
```

```
 # Example usage
if __name__ == "__main__":
    # Example distance matrix (replace with your own)
    distances = np.array([[0, 10, 15, 20],
                          [10, 0, 35, 25],
                          [15, 35, 0, 30],
                          [20, 25, 30, 0]])
   # Solve TSP using nearest neighbor heuristic
    tour = tsp_nearest_neighbor(distances)
    print("Tour:", tour)
```

output

Tour: [0, 1, 3, 2, 0]

7. Implementation of the problem solving strategies: either using Forward Chaining or Backward Chaining

```
class KnowledgeBase:

    def __init__(self):

        self.facts = set()

        self.rules = []

   def tell_fact(self, fact):

        self.facts.add(fact)

   def tell_rule(self, antecedent, consequent):

        self.rules.append((antecedent, consequent))

  def forward_chaining(self, goal):

        inferred_facts = set()

        agenda = []

        # Initialize agenda with known facts

        agenda.extend(self.facts)

       # Repeat until the goal is reached or the agenda is empty

        while agenda:

            current_fact = agenda.pop(0)

   # If the current fact matches the goal, return True

            if current_fact == goal:

                return True

 # Check rules for inference

            for antecedent, consequent in self.rules:
```

```
                if antecedent in inferred_facts:
                    if consequent not in inferred_facts and consequent not in agenda:
                        agenda.append(consequent)
            # Add current fact to inferred facts
            inferred_facts.add(current_fact)
        # Goal not reached
        return False
# Example usage
if __name__ == "__main__":
    kb = KnowledgeBase()
    # Define facts
    kb.tell_fact("A")
    kb.tell_fact("B")
    # Define rules
    kb.tell_rule("A", "C")
    kb.tell_rule("B", "C")
    kb.tell_rule("C", "D")
    # Perform forward chaining
    goal = "D"
    if kb.forward_chaining(goal):
        print(f"The goal '{goal}' is reachable.")
    else:
        print(f"The goal '{goal}' is not reachable.")
```

output

The goal 'D' is reachable.

8. Implement resolution principle on FOPL related problems

```
class Predicate:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __eq__(self, other):
        return isinstance(other, Predicate) and self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash((self.name, tuple(self.args)))
    def __str__(self):
        return f"{self.name}({', '.join(self.args)})"
class Clause:
    def __init__(self, literals):
        self.literals = set(literals)
    def __eq__(self, other):
        return isinstance(other, Clause) and self.literals == other.literals
    def __hash__(self):
        return hash(tuple(sorted(self.literals)))
    def __str__(self):
        return " | ".join(str(lit) for lit in self.literals)
def resolve(clause1, clause2):
    resolvents = set()
    for lit1 in clause1.literals:
        for lit2 in clause2.literals:
            if lit1.name == lit2.name and lit1.args != lit2.args:
                new_clause_literals = (clause1.literals | clause2.literals) - {lit1, lit2}
                new_clause = Clause(new_clause_literals)
```

```
            resolvents.add(new_clause)
    return resolvents
def resolve_algorithm(knowledge_base, query):
    resolved = False
    new_clauses = set()
    while not resolved:
        resolved = True
        for clause1 in knowledge_base:
            for clause2 in knowledge_base:
                if clause1 != clause2:
                    resolvents = resolve(clause1, clause2)
                    for resolvent in resolvents:
                        if resolvent not in knowledge_base and resolvent not in new_clauses:
                            new_clauses.add(resolvent)
                            resolved = False
                            if query in resolvent.literals:
                                return True
        knowledge_base.update(new_clauses)
    return False
# Example usage
if __name__ == "__main__":
    # Define knowledge base
    knowledge_base = {
        Clause({Predicate("P", ["a", "b"]), Predicate("Q", ["a"])}),
        Clause({Predicate("P", ["x", "y"])}),
        Clause({Predicate("Q", ["y"]), Predicate("R", ["y"])}),
        Clause({Predicate("R", ["z"])}),
    }
```

```
# Define query
query = Predicate("R", ["a"])
# Perform resolution
result = resolve_algorithm(knowledge_base, query)
if result:
    print("Query is satisfiable.")
else:
    print("Query is unsatisfiable.")
```

output

The given set of clauses is satisfiable.

9. Implement Tic-Tac-Toe game using Python

```
class TicTacToe:
  def __init__(self):
      self.board = [' ' for _ in range(9)]
      self.current_player = 'X'
  def print_board(self):
      for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
          print('| ' + ' | '.join(row) + ' |')
  def make_move(self, position):
      if self.board[position] == ' ':
          self.board[position] = self.current_player
          if self.check_winner(position):
              print(f"Player {self.current_player} wins!")
              return True
          elif ' ' not in self.board:
              print("It's a tie!")
              return True
          else:
              self.current_player = 'O' if self.current_player == 'X' else 'X'
          return False
      else:
          print("That position is already taken!")
          return False
   def check_winner(self, position):
      row_index = position // 3
      col_index = position % 3
```

```
        # Check row
       if all(self.board[row_index*3 + i] == self.current_player for i in range(3)):
            return True
        # Check column
        if all(self.board[col_index + i*3] == self.current_player for i in range(3)):
            return True
      # Check diagonal
        if row_index == col_index and all(self.board[i*3 + i] == self.current_player for i in
range(3)):
            return True
    # Check anti-diagonal
        if row_index + col_index == 2 and all(self.board[i*3 + (2-i)] == self.current_player for i in
range(3)):
            return True
          return False
 def main():
    game = TicTacToe()
    while True:
        game.print_board()
        position = int(input(f"Player {game.current_player}, enter your position (0-8): "))
        if game.make_move(position):
            break
if __name__ == "__main__":
main()
```

output

```
|  |  |  |
```

```
|   |   |   |
|   |   |   |
Player X, enter your position (0-8): 0
| X |   |   |
|   |   |   |
|   |   |   |
Player O, enter your position (0-8): 1
| X | O |   |
|   |   |   |
|   |   |   |
Player X, enter your position (0-8): 3
| X | O |   |
| X |   |   |
|   |   |   |
Player O, enter your position (0-8): 4
| X | O |   |
| X | O |   |
|   |   |   |
Player X, enter your position (0-8): 6
Player X wins!
```

10. Implement any Game and demonstrate the Game playing strategies

```
import random
class RockPaperScissors:
    def __init__(self):
        self.moves = ['rock', 'paper', 'scissors']
        self.winning_moves = {'rock': 'scissors', 'paper': 'rock', 'scissors': 'paper'}
    def play_round(self, player_move):
        computer_move = random.choice(self.moves)
        print(f"Computer chooses: {computer_move}")
        if player_move == computer_move:
            return "It's a tie!"
        elif self.winning_moves[player_move] == computer_move:
            return "You win!"
        else:
            return "Computer wins!"
class RandomPlayer:
    def __init__(self):
        pass
def get_move(self):
        return random.choice(['rock', 'paper', 'scissors'])
class AIPlayer:
    def __init__(self):
        pass
def get_move(self, player_moves):
        # Basic strategy: Counter the player's last move
        last_move = player_moves[-1]
        if last_move == 'rock':
```

```
            return 'paper'
        elif last_move == 'paper':
            return 'scissors'
        else:
            return 'rock'

# Demonstration
if __name__ == '__main__':
    game = RockPaperScissors()
    player1 = RandomPlayer()
    player2 = AIPlayer()
    player_moves = []
   for _ in range(3):
        # Random player's move
        player_move = player1.get_move()
        player_moves.append(player_move)
        print(f"You chose: {player_move}")
        # AI player's move
        ai_move = player2.get_move(player_moves)
        player_moves.append(ai_move)
        print(f"AI chose: {ai_move}")
        # Determine the winner of the round
        result = game.play_round(player_move)
        print(result)
```

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

Output

You chose: rock

AI chose: paper

Computer chooses: scissors

You win!

You chose: rock

AI chose: paper

Computer chooses: paper

Computer wins!

You chose: scissors

AI chose: rock

Computer chooses: paper

You win!
