

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | **2117240070028** |
| **NAME** | **Ashwitha R** |
| **PROJECT TITLE** | **ROCK PAPER SCISSORS USING MONTE CARLO TREE SEARCH** |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION:**

Artificial Intelligence (AI) techniques are widely used to simulate intelligent decision-making in games and real-world problems. One such method is Monte Carlo Tree Search (MCTS) — a powerful algorithm that uses random sampling of future game outcomes to select the best move.

In this project, we develop a Rock Paper Scissors (RPS) game where the computer learns and adapts using MCTS. Unlike static rule-based AIs, MCTS explores multiple possible moves, simulates outcomes, and updates its strategy dynamically based on results. This demonstrates how probabilistic reasoning can improve AI decisions even in simple games.

**PROBLEM STATEMENT:**

To design and implement an AI agent for the Rock Paper Scissors game using Monte Carlo Tree Search, where the AI predicts the best move through repeated simulations and statistical analysis instead of random guessing.

**GOAL:**

Develop an adaptive RPS game using MCTS.

Apply Monte Carlo simulations to predict opponent patterns.

Compare win probabilities and choose the best action (Rock, Paper, or Scissors).

Demonstrate AI decision-making using probabilistic search instead of fixed rules.

**THEORETICAL BACKGROUND:**

Monte Carlo Tree Search (MCTS):

MCTS is a heuristic search algorithm that makes decisions by simulating random plays and updating statistics for each action. It consists of four key steps:

1. Selection: Traverse the tree from the root using a selection policy (like UCT – Upper Confidence Bound).

2. Expansion: Add a new node representing an unvisited move.

3. Simulation (Rollout): Play random games until reaching an outcome (win, loss, or draw).

4. Backpropagation: Update node statistics (wins and visits) based on the result.

Over multiple iterations, the algorithm converges toward the most promising move.

**ALGORITHM EXPLANATION WITH EXAMPLE:**

Steps:

1. Start from the current game state.

2. Generate all possible moves: Rock, Paper, Scissors.

3. Run multiple random simulations for each move to estimate outcomes.

4. Record results (wins, losses, draws).

5. Select the move with the highest win ratio.

6. Play the move, update the tree, and repeat for the next round.

Example:

Suppose AI considers the following after 100 simulations:

Rock → 40 wins

Paper → 30 wins

Scissors → 30 wins

The AI will choose Rock, as it has the highest win probability (40%).

**IMPLEMENTATION AND CODE:**

import random

import math

import time

# Possible moves

MOVES = ["Rock", "Paper", "Scissors"]

# Function to decide the winner

def get\_winner(player, opponent):

if player == opponent:

return 0 # Draw

elif (player == "Rock" and opponent == "Scissors") or \

(player == "Paper" and opponent == "Rock") or \

(player == "Scissors" and opponent == "Paper"):

return 1 # Win

else:

return -1 # Loss

# Monte Carlo Tree Search for RPS

def mcts(num\_simulations=1000):

results = {move: {"wins": 0, "plays": 0} for move in MOVES}

for move in MOVES:

for \_ in range(num\_simulations):

opponent = random.choice(MOVES)

result = get\_winner(move, opponent)

results[move]["plays"] += 1

if result == 1:

results[move]["wins"] += 1

# Calculate win rate

best\_move = max(MOVES, key=lambda m: results[m]["wins"] / results[m]["plays"])

return best\_move, results

# Game loop

def play\_game():

print("Rock Paper Scissors using Monte Carlo Tree Search")

while True:

user = input("Enter your move (Rock/Paper/Scissors or Quit): ").capitalize()

if user.lower() == "quit":

print("Game Over!")

break

if user not in MOVES:

print("Invalid move! Try again.")

continue

start = time.time()

ai\_move, stats = mcts(num\_simulations=500)

elapsed = time.time() - start

print(f"AI chose: {ai\_move}")

result = get\_winner(user, ai\_move)

if result == 1:

print("You win!")

elif result == -1:

print("AI wins!")

else:

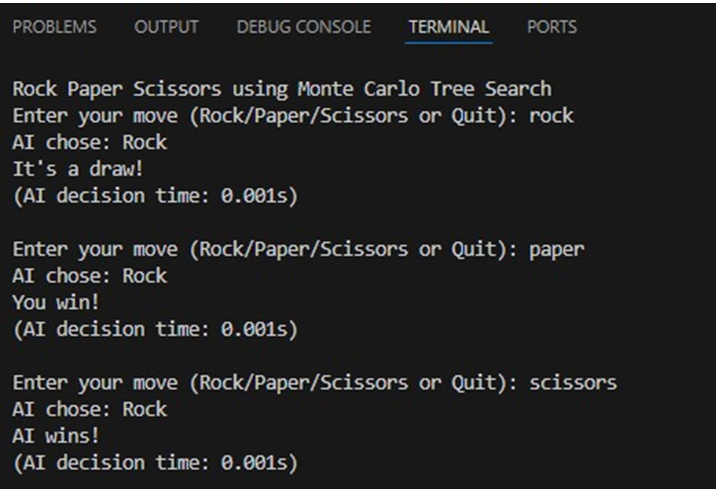
print("It's a draw!")

print(f"(AI decision time: {elapsed:.3f}s)\n")

if \_name\_ == "\_main\_":

play\_game()

**OUTPUT:**



**RESULTS AND FUTURE ENHANCEMENT:**

Results:  
The MCTS-based AI can intelligently select moves with higher win probabilities.  
Increasing the number of simulations improves accuracy but increases computation time.  
Demonstrates how MCTS can apply to even simple, non-deterministic games.

Future Enhancements:

Incorporate opponent pattern learning (track user history).

Use weighted simulations based on previous outcomes.

Add a graphical user interface for better visualization.

Implement parallelized MCTS for faster results.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | https://github.com/ASHWITHA2202/AI-mini-project |

**REFERENCES:**

Russell, S. J., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th Edition). Pearson Education.

Browne, C., et al. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Transactions on Computational Intelligence and AI in Games.

GeeksforGeeks – “Monte Carlo Tree Search in Python.”