Experiment No. 4

Title:

Min Max algorithm in Game Theory Program using Python.

Aim:

The aim of this lab is to implement the Minimax algorithm, a fundamental concept in game theory, using Python. Students will gain practical experience in coding an algorithm that helps in decision-making for two-player games with perfect information.

Objectives:

1. To understand the fundamental concepts of Min-Max Algorithm.
2. To implement the Min-Max algorithm using Python.

Problem Statement:

Implementation of Min-Max Algorithm for Game Decision Making.

Software/s: Python (version 3.x)

Theory:

Minimax Algorithm:

The Minimax algorithm is commonly used in two-player games with perfect information, such as tic-tac-toe or chess. It is a decision-making algorithm that aims to minimize the possible loss for a worst-case scenario while maximizing potential gains. The algorithm assumes that both players are playing optimally.

Algorithm Steps:

- Initialization:
  - Begin with the current state of the game.
- Generate Game Tree:
  - Generate the entire game tree, representing all possible moves and resulting states up to a certain depth.
- Evaluation Function:
  - Apply an evaluation function to assign a score to each terminal state (end of the game).
- Backpropagation:
  - Propagate scores upward through the tree, alternating between minimizing and maximizing players.

- Decision-Making:
  - Make the move that leads to the path with the highest score when playing as the maximizing player or the lowest score when playing as the minimizing player.

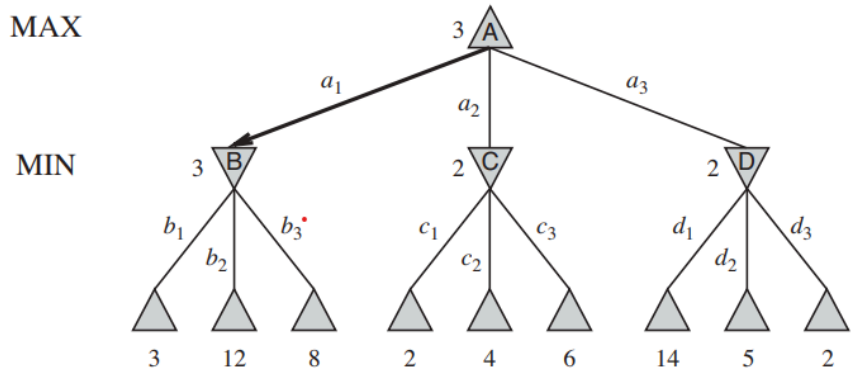

Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. For example, in Figure 5.2, the algorithm first recurses down to the three bottom left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(b m). The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time (see page 87). For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Procedure:

1. Game Representation:
   - Define a representation for the game state, considering the current positions, scores, and possible moves.
2. Minimax Algorithm Implementation:
   - Write a Python script to implement the Minimax algorithm.
   - Develop functions for generating the game tree, applying the evaluation function, and making decisions.
3. Game Implementation:
   - Choose a simple two-player game (e.g., tic-tac-toe) to test the Minimax algorithm.
4. Visualization (Optional):
   - Optionally, visualize the decision-making process using a graphical representation of the game tree.
5. Experimentation:
   - Experiment with different game scenarios to observe how the Minimax algorithm makes optimal decisions.
6. Complexity and Optimization:
   - Analyze the time and space complexity of the Minimax algorithm.
   - Explore optimization techniques, such as alpha-beta pruning, to improve performance.

Conclusion:

_____
_____
_____
_____
_____

References:

S. Russel, P. Norvig, “Artificial Intelligence – A Modern Approach”, Third Edition, Pearson Education, 2015.