Lab 7

29/07/2024

- ## Alpha-Beta Pruning

Summary

The provided code implements the Minimax algorithm with alpha-beta pruning. The Minimax algorithm is used for decision-making in games to find the optimal move for a player, assuming the opponent also plays optimally. Alpha-beta pruning is an optimization technique that reduces the number of nodes evaluated by the Minimax algorithm by pruning branches that cannot possibly influence the final decision.

Workflow

1. **Initialization:**
    - MAX and MIN are set to 1000 and -1000 respectively, representing initial extreme values.
    - The minimax function is defined to recursively evaluate the game tree.
    - A list values contains terminal values of the game tree at depth 3.
2. **Minimax Function:**
    - Base Case: If depth equals 3, the function returns the value at nodeIndex in values.
    - Maximizing Player: If it's the maximizing player's turn:
        - Initialize best to MIN.
        - Recursively call minimax for the next depth and the two possible child nodes.
        - Update best to the maximum value obtained.
        - Update alpha to the maximum of alpha and best.

        - If beta is less than or equal to alpha, prune the branch by breaking out of the loop.
    - Minimizing Player: If it's the minimizing player's turn:
        - Initialize best to MAX.
        - Recursively call minimax for the next depth and the two possible child nodes.
        - Update best to the minimum value obtained.
        - Update beta to the minimum of beta and best.
        - If beta is less than or equal to alpha, prune the branch by breaking out of the loop.

3. **Execution:**
    - Call minimax with initial parameters (depth=0, nodeIndex=0, maximizingPlayer=True, values, alpha=MIN, beta=MAX).
    - Print the optimal value obtained from the Minimax function.

Explanation

1. **Minimax Algorithm:**
    - Purpose: To find the optimal move for a player assuming the opponent also plays optimally.
    - Mechanism: The algorithm constructs a game tree, evaluates possible moves, and chooses the best move for the maximizing player and the worst for the minimizing player.
2. **Alpha-Beta Pruning:**
    - Purpose: To optimize the Minimax algorithm by reducing the number of nodes evaluated.
    - Mechanism: alpha represents the best value the maximizing player can guarantee, while beta represents the best value the minimizing player can guarantee. If at any point beta is less than or equal to alpha, further evaluation of that branch is stopped (pruned).

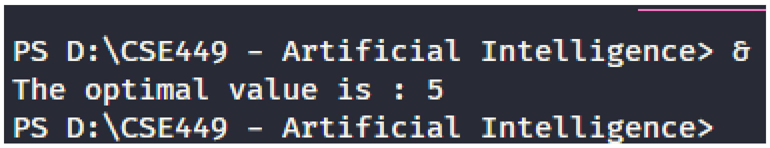
3. **Example Execution:**

   - The function starts at depth 0, node index 0, and with the maximizing player.
   - It recursively evaluates the game tree, alternating between maximizing and minimizing players.
   - Alpha-beta pruning reduces unnecessary evaluations.
   - Finally, it returns the optimal value based on the given values list.

Code:

```
MAX, MIN = 1000, -1000


def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]


    if maximizingPlayer:
        best = MIN
        for i in range(0, 2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)


            if beta <= alpha:
                break
        return best
    else:

        best = MAX

        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)

            best = min(best, val)

            beta = min(beta, best)


            if beta <= alpha:

                break

        return best


values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:



```
PS D:\CSE449 - Artificial Intelligence> &
The optimal value is : 5
PS D:\CSE449 - Artificial Intelligence>
```

• Tower of Hanoi:

Summary

The Tower of Hanoi is a classic problem in computer science and mathematics. The objective is to move a stack of disks from one rod to another, following certain rules:

1. Only one disk can be moved at a time.
2. Each move involves taking the top disk from one of the stacks and placing it on top of another stack or on an empty rod.

3. No disk may be placed on top of a smaller disk.

Workflow

1. Initialization:
   - Define the function TowerOfHanoi which takes four parameters: the number of disks n, the source rod, the destination rod, and the auxiliary rod.
   - If there is only one disk, move it directly from the source to the destination.
2. Recursive Approach:
   - Move n-1 disks from the source rod to the auxiliary rod using the destination rod.
   - Move the nth disk from the source rod to the destination rod.
   - Move the n-1 disks from the auxiliary rod to the destination rod using the source rod.
3. Execution:
   - Call the TowerOfHanoi function with the initial parameters to solve the problem for n disks.

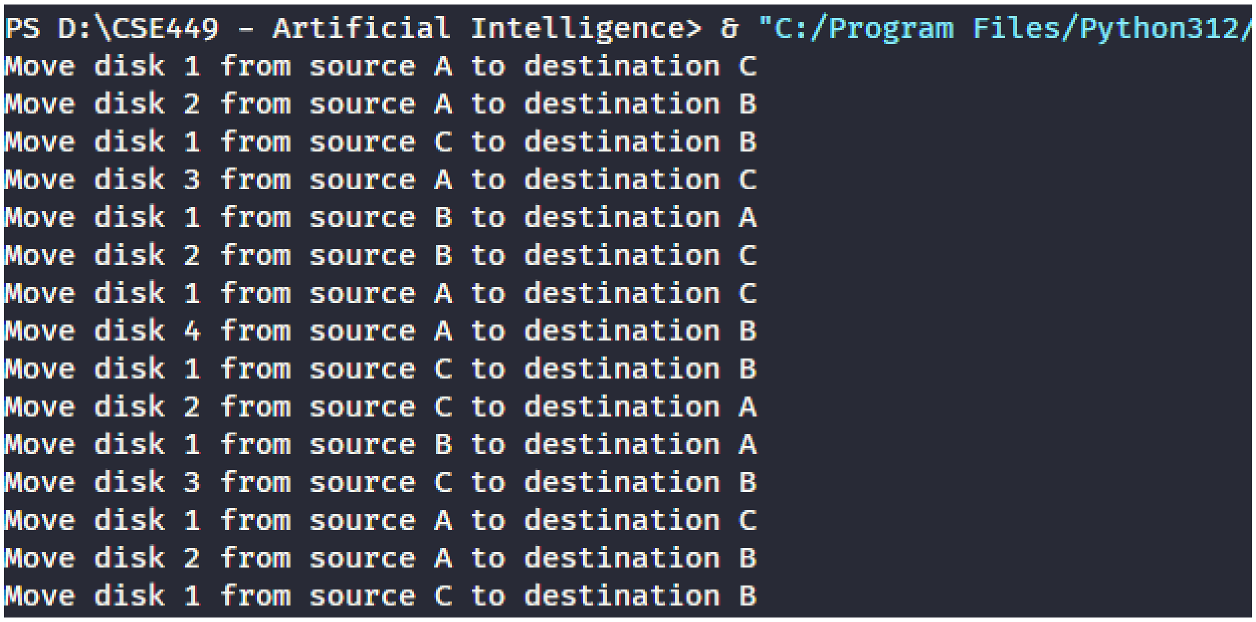
Explanation

1. Base Case:
   - If n is 1, the function prints a message to move the disk from the source to the destination and returns.
2. Recursive Case:
   - Move n-1 disks from the source to the auxiliary rod using the destination rod.
   - Print the move of the nth disk from the source to the destination.
   - Move n-1 disks from the auxiliary rod to the destination rod using the source rod.
3. Example Execution:
   - For n = 4, the function will print the sequence of moves needed to transfer all disks from rod "A" to rod "B" using rod "C" as the auxiliary rod.

Code:

```
def TowerOfHanoi(n, source, destination, auxiliary):

    if n == 1:

        print(f"Move disk 1 from source {source} to destination {destination}")

        return

    TowerOfHanoi(n - 1, source, auxiliary, destination)
    print(f"Move disk {n} from source {source} to destination {destination}")

    TowerOfHanoi(n - 1, auxiliary, destination, source)



n = 4

TowerOfHanoi(n, "A", "B", "C")
```

Output:



```
PS D:\CSE449 - Artificial Intelligence> & "C:/Program Files/Python312/
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```

- ## Water Jug Problem:

Summary

The water jug problem is a classic problem in which you have two jugs of different capacities and you need to measure a specific amount of water using these jugs. Here, you have two jugs with capacities of 4 liters and 3 liters, and the goal is to measure exactly 2 liters.

Workflow

1. **Initialization**:
    - Define the capacities of the two jugs (jug1 and jug2).
    - Define the target amount (aim).

- Use a defaultdict to keep track of visited states to avoid redundant calculations.

2. **Recursive Solver**:
   - Define the function waterJugSolver which takes the current amount of water in both jugs (amt1 and amt2).
   - Base Case: If either jug contains the target amount (aim) and the other jug is empty, print the current state and return True.
   - Recursive Case: If the current state has not been visited, mark it as visited and explore all possible moves:
     - Empty either jug.
     - Fill either jug.
     - Pour water from one jug to the other until one is either full or the other is empty.
3. **Execution**:
   - Call the waterJugSolver function with initial amounts (0, 0).

Explanation

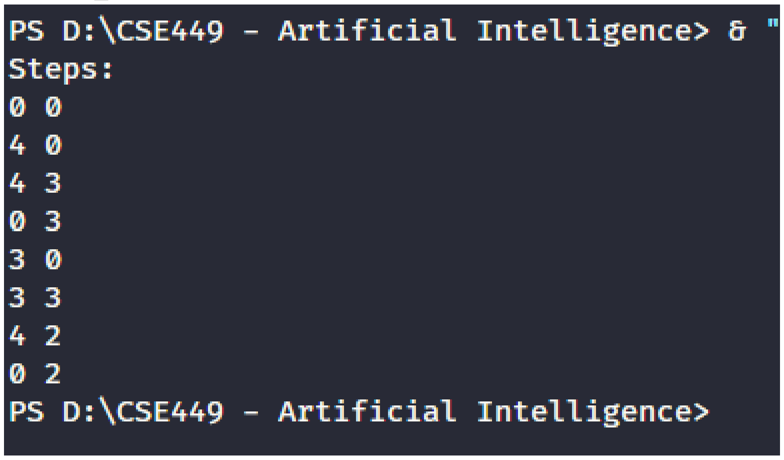
1. **Base Case**:
   - If one jug has the aim amount and the other is empty, print the amounts and return True.
2. **Recursive Case**:
   - If the current state has not been visited, mark it as visited.
   - Explore the six possible moves:
     1. Empty jug1.
     2. Empty jug2.
     3. Fill jug1.
     4. Fill jug2.
     5. Pour water from jug2 to jug1.
     6. Pour water from jug1 to jug2.
   - If any move returns True, propagate the True value upwards.
3. **Example Execution**:
   - For jug1 = 4, jug2 = 3, and aim = 2, the function will print the sequence of steps needed to measure exactly 2 liters.

Code:

```
from collections import defaultdict
```

```
jug1, jug2, aim = 4, 3, 2


visited = defaultdict(lambda: False)


def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True


    if not visited[(amt1, amt2)]:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True


        return (
            waterJugSolver(0, amt2)
            or waterJugSolver(amt1, 0)
            or waterJugSolver(jug1, amt2)
            or waterJugSolver(amt1, jug2)
            or waterJugSolver(
                amt1 + min(amt2, (jug1 - amt1)), amt2 - min(amt2, (jug1 - amt1))
            )
            or waterJugSolver(
                amt1 - min(amt1, (jug2 - amt2)), amt2 + min(amt1, (jug2 - amt2))
            )
        )
    else:
        return False
print("Steps:")
waterJugSolver(0, 0)
```

Output:



```
PS D:\CSE449 - Artificial Intelligence> & "
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
PS D:\CSE449 - Artificial Intelligence>
```