🔵 **AI Practical 3 – Minimax Algorithm with Alpha-Beta Pruning**

🟣 **1. THEORY (Explain to External)**

✅ What is Minimax Algorithm?

Minimax is a **game-playing search algorithm** used in:

- Tic-Tac-Toe
- Chess
- Checkers
- Connect 4
- Any turn-based 2-player game

It tries to **maximize the player's score** and **minimize opponent's score**.

🧠 How Minimax Works?

- Two players:
  MAX player → tries to get highest score
  MIN player → tries to get lowest score
- Minimax explores **all possible moves** in a game tree until leaf nodes.

🔥 What is Alpha-Beta Pruning?

Alpha-Beta Pruning is an optimization technique that:

- **Cuts unnecessary branches** of the game tree
- Reduces computation
- Speeds up Minimax
- Does NOT affect the final answer (optimal value same)

💡 Key Terms

- **Alpha (α)** → best value found for MAX so far

- **Beta (β)** → best value found for MIN so far
- **Pruning** occurs when:
- beta <= alpha

Meaning: further evaluation is useless → stop searching that branch.

🟢 2. FULL CODE (Write This Exact Code in Practical)

```
MAX = 1000

MIN = -1000


def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

  if depth == 3:

    return values[nodeIndex]


  if maximizingPlayer:

    best = MIN

    for i in range(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(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


if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

🔵 3. LINE-BY-LINE EXPLANATION (VERY EASY & CLEAR)

📍 Initialization of MAX & MIN

```
MAX = 1000
MIN = -1000
```

We set:

- MAX = 1000 → a very large value
- MIN = -1000 → a very small value
  Used to initialize best values.

📌 Minimax Function

```
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
```

This function performs minimax search with alpha-beta pruning.

✔️ Parameters:

- depth → current level in tree
- nodeIndex → index of value in array (leaf node)
- maximizingPlayer → True (MAX), False (MIN)

- values → array storing terminal values
- alpha → best score for MAX
- beta → best score for MIN

📍 Base Condition

```
if depth == 3:
    return values[nodeIndex]
```

When depth reaches 3 → this is a leaf node → return the leaf value.

📗 Case 1: Maximizing Player (MAX)

```
if maximizingPlayer:
    best = MIN
```

Start from lowest possible value.

Loop through children

```
    for i in range(2):
```

Each node has **2 children**.

Recursively call minimax for children

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

- Increase depth
- Move to child node (formula: nodeIndex * 2 + i)
- Next player is MIN (False)

Update best for MAX

```
        best = max(best, val)
```

Choose the maximum child value.

Update α for MAX

```
alpha = max(alpha, best)
```

Check for pruning

```
if beta <= alpha:
    break
```

If beta ≤ alpha → stop exploring remaining children (pruning).

📗 Case 2: Minimizing Player (MIN)

```
else:
    best = MAX
```

Start from highest possible value.

Loop through children

```
for i in range(2):
```

Recursive call for MAX player

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

MIN chooses minimum

```
best = min(best, val)
```

Update β for MIN

```
beta = min(beta, best)
```

Pruning

```
if beta <= alpha:
    break
```

📌 Return Best Value

return best

📘 Running Code

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

These are leaf node values of the game tree.

Execute minimax

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

- Start from root
- Depth = 0
- MAX player starts

🎉 Final Output

The optimal value is: 5

Line-by-line — every word, every symbol explained (so even a class-9 student can read it to the teacher)

MAX = 1000

MIN = -1000

- MAX — a name (variable). We use it to mean a **very large positive number**.
- = — assignment operator. "Set the name on the left to the value on the right."
- 1000 — the number one thousand. Here it means "a very big positive value."
- MIN — another name (variable). We use it to mean a **very small (negative) number**.
- -1000 — minus one thousand, a big negative number.
  Why: Later we start comparisons from these extremes. MAX is used as starting "worst for MIN" and MIN as starting "worst for MAX."

---

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

- def — keyword that starts a **function definition**. It says “I am defining a function.”
- minimax — the function’s name. This is the label you will call to run the code inside.
- ( ) — parentheses hold the input names (parameters) the function accepts.
- depth — parameter name. It tells which level of the tree we are at (0 for root, 1 next, etc.).
- , — separates parameters.
- nodeIndex — parameter name. It tells which leaf index in the values list corresponds to the current node path.
- maximizingPlayer — parameter name. This is a True/False flag that says whose turn it is: True means MAX's turn, False means MIN's turn.
- values — parameter name. This is a list (array) of numbers representing scores at the leaf nodes of the full tree.
- alpha — parameter name. It stores the best value (highest) found so far for MAX while searching.
- beta — parameter name. It stores the best value (lowest) found so far for MIN while searching.
- : — colon marks the start of the function body (indented block below).

Plain: This line declares a function called minimax that will take six pieces of information and do calculations.

---

  if depth == 3:

- if — keyword for a conditional. It says “do the next block only when the condition is true.”
- depth — the current depth number passed into the function.
- == — equality comparison operator. Checks if left and right are exactly equal.
- 3 — the number three. Here it means “we stop after 3 levels.”
- : — colon starts the block that runs when the condition is true.

Plain: If we have gone down 3 levels, we are at a leaf node — do the next line.

```
    return values[nodeIndex]
```

- return — keyword that sends a result back to the caller and stops the function.
- values — the list of leaf numbers you gave.
- [ ] — square brackets are used to pick one element from a list by index.
- nodeIndex — the index (position) in the values list to pick. Indices start from 0.
  Plain: Give back the value at position nodeIndex in the values list. This is the score when we reach a leaf.

```
  if maximizingPlayer:
```

- if — conditional again.
- maximizingPlayer — the boolean flag parameter. If it's True, we enter this block.
- : — start of the block for MAX's turn.

Plain: If it's MAX's turn, do the following steps.

```
    best = MIN
```

- best — a new variable name inside the function, used to store the best value MAX can get from this node so far.
- = — assign operator.
- MIN — the negative big number defined earlier.
  Plain: Start best with a very small value so any real child value will be bigger.

```
    for i in range(2):
```

- for — loop keyword. It repeats the indented block below for each item in a sequence.
- i — loop variable. It will take values 0 and then 1.
- in — part of the loop syntax — “for i that is in ...”
- range(2) — generate sequence 0, 1. range(2) means two iterations.

- : — starts the loop block.

Plain: Repeat the block for each of the two children of this node (binary tree).

---

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

- val — name for a variable to store the result returned by the recursive call.
- = — assign.
- minimax(...) — calling the minimax function again (recursion). Inside (...) we provide new arguments:
    - depth + 1 — take current depth and add 1 (we go one level deeper).
        - + — addition operator.
    - nodeIndex * 2 + i — calculate the child's index in the leaf values list:
        - nodeIndex * 2 doubles the index (left child index formula).
        - + i adds 0 for left child, 1 for right child.
        - * and + are arithmetic operators.
    - False — pass boolean False. This means after MAX moves, the next player is MIN.
    - values — pass the same list of leaf values.
    - alpha — pass current alpha value unchanged.
    - beta — pass current beta value unchanged.
      Plain: Ask the function to find the value of the child node and store it in val.

---

best = max(best, val)

- best — current best value for MAX so far.
- = — assign.
- max(best, val) — built-in function max returns the bigger of the two numbers best and val.
  Plain: Update best to the higher of existing best and the child's val.

---

alpha = max(alpha, best)

- alpha — the current best value for MAX saved in the alpha variable.
- max(alpha, best) — pick the larger of the old alpha and updated best.
  Plain: Update alpha to reflect the best value MAX can guarantee now.

```
    # Alpha-Beta pruning
```

- # — starts a comment. Everything after # on the same line is ignored by Python and for humans only.
- Alpha-Beta pruning — note to say this part does pruning.

Plain: This is a comment labeling the next check.

```
    if beta <= alpha:
```

- if — conditional.
- beta — current best value for MIN.
- <= — less-than-or-equal comparison operator.
- alpha — current best for MAX.
  Plain: If MIN's best so far is less than or equal to MAX's best so far, then no need to search more at this node.

```
     break
```

- break — keyword that immediately exits the nearest loop (for loop here).
  Plain: Stop checking more children because further children cannot change the decision (prune).

```
  return best
```

- return — sends best value back to the caller.
- best — the chosen maximal value for this node after considering children (and pruning).
  Plain: Give MAX’s final best value for this position.

```
else:
```

- else — attaches to the if maximizingPlayer above. This block runs when maximizingPlayer is False (i.e., it’s MIN’s turn).
- : — starts the else block.

Plain: If it is NOT MAX's turn, do the MIN logic below.

```
best = MAX
```

- best — new variable for MIN inside this branch.
- = — assign.
- MAX — the big positive number we set earlier.
  Plain: MIN starts with a very large number; MIN will look for smaller numbers (worse for MAX).

```
for i in range(2):
```

- same as before: loop over two children (0 and 1).

Plain: MIN will check both children unless pruning stops it.

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

- same as the MAX call, but now True is passed so the next player will be MAX.
  Plain: Ask for the value of child node when it becomes MAX's turn.

```
  best = min(best, val)
```

- min(best, val) — built-in min returns the smaller number.
  Plain: MIN updates its best to the smaller value (MIN wants to minimize the score).

```
  beta = min(beta, best)
```

- update beta to the smallest between the old beta and best.
  Plain: Beta stores the best (lowest) value MIN can force so far.

```
  # Alpha-Beta pruning
```

- comment again marking pruning check.

```
    if beta <= alpha:
```

- same comparison as earlier. If MIN’s best (beta) is <= MAX’s best (alpha), the rest of the siblings cannot help MIN produce a better result for itself, so skip them.

Plain: If MIN now has a value that is already worse for MAX than MAX’s current best, stop exploring.

```
        break
```

- break out of the for loop for children.
  Plain: Prune remaining siblings.

```
    return best
```

- return the chosen minimal value for this MIN node.

Plain: Give MIN’s final best value upward.

```
if __name__ == "__main__":
```

- if — conditional.
- __name__ — a built-in special variable in Python that says how the file is being run.
- == — equality check.
- "__main__" — a special string. When a Python file is run directly (not imported), __name__ equals "__main__".
- : — start block.

Plain: This block runs only when you run this file directly (not when you import it from another file). It’s standard for code that should run as a script.

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

- values — name for the list of leaf scores.
- = — assign.

- [ ] — list literal.
- 3, 5, 6, 9, 1, 2, 0, -1 — eight numbers inside the list. They represent the final scores at the leaves (leftmost leaf index 0 to rightmost leaf index 7).
  Plain: These are the outcomes at the bottom of the game tree.

---

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

- print — built-in function that outputs text to the screen.
- ("The optimal value is:", ...) — two arguments separated by a comma:
  - "The optimal value is:" — a string to show before the number.
  - minimax(0, 0, True, values, MIN, MAX) — call the minimax function with:
    - 0 → starting depth (root level).
    - 0 → starting nodeIndex (root index).
    - True → start with MAX's turn.
    - values → the list we defined.
    - MIN → initial alpha (very small).
    - MAX → initial beta (very large).
-) — closes the print call.

Plain: Run the minimax algorithm from the top of the tree and print the result with a descriptive message.

Final simple story you can tell your teacher (one-sentence):

“The program builds a two-player game search: MAX tries to maximize score, MIN tries to minimize it; we search all leaf scores using recursion but skip whole parts of the tree early (alpha–beta pruning) when we can prove they won’t change the outcome, and the code returns the best possible value MAX can force, which here is printed at the end.”

Extra — What each function call and symbol does when running (step flow in plain language)

1. The script sets MAX and MIN anchors.
2. values list contains leaf outcomes.

3. minimax(0, 0, True, values, MIN, MAX) starts at the root.
4. At each level depth, the function:
    - If depth == 3: return the leaf value (stop).
    - If maximizingPlayer == True: try both children, ask recursively for their values, keep the maximum, update alpha, prune children if beta <= alpha.
    - If maximizingPlayer == False: try both children similarly, keep minimum, update beta, prune if beta <= alpha.
5. Recursion unwinds returning the chosen optimal values upward until the root returns the best value for MAX.
6. print outputs that best value.

Viva (questions your teacher may ask) — with short ready-to-say answers

1. **Q: What does minimax do?**
   A: It explores the game tree to decide the best outcome assuming both players play optimally — MAX maximizes and MIN minimizes.
2. **Q: Why use alpha and beta?**
   A: To cut (prune) branches that cannot affect the final result, which makes search faster.
3. **Q: Why depth == 3?**
   A: That is the tree height chosen in this example — leaves are at depth 3. If tree size changed, we would change this number.
4. **Q: What is nodeIndex * 2 + i?**
   A: It's the formula to compute the child's index in a full binary tree when leaves are stored in an array.
5. **Q: Why best = MIN for MAX and best = MAX for MIN?**
   A: We initialize best to the worst possible case for the player so that any real candidate will improve it.
6. **Q: What does break do here?**
   A: It stops checking further children of the node because they are unnecessary (pruned).
7. **Q: If you removed alpha-beta pruning (the alpha, beta, and if checks), would the result change?**
   A: No — the final optimal value remains the same. Only the performance (time) changes.

8. **Q: Why pass True and False alternately?**
   A: To switch turns between MAX and MIN down the tree.
9. **Q: What would happen if we had 3 children per node instead of 2?**
   A: The for loop would change to range(3) and the nodeIndex child formula would change accordingly (for array layout you'd adapt indexing).
10. **Q: Why are MAX and MIN numbers like 1000 and -1000 and not infinity?**
    A: Using a large sentinel covers practical ranges. We could use float('inf') and float('-inf') for true infinity, but these numbers work for this example.

Quick practice script for you to say to the teacher (30–40 seconds)

"First the program defines very big and very small numbers. Then it defines a minimax function which, given a node, depth and whose turn it is, returns the optimal leaf value reachable from that node. If we are at leaf level (depth 3) it returns the leaf directly. If it's MAX's turn, it takes the largest value from children, updates alpha, and stops early if beta <= alpha. If it's MIN's turn, it takes the smallest child, updates beta, and also prunes when beta <= alpha. Finally we run the function from the root and print the optimal value."

🔥 4. VIVA QUESTIONS (EXAM-READY)

---

1. What is Minimax Algorithm?

Minimax is a game tree search algorithm used for decision making in two-player games.

---

2. What is the objective of MAX player?

MAX player tries to get the **highest score**.

---

3. What is the objective of MIN player?

MIN player tries to get the **lowest score**.

---

4. What is Alpha-Beta Pruning?

A technique to **stop evaluating unnecessary branches** in minimax tree.

---

5. Does Alpha-Beta change the final result?

No.
It only reduces time, output remains the same.

---

6. What is Alpha and Beta?

- **Alpha** = best value for MAX
- **Beta** = best value for MIN

---

7. When does pruning happen?

When:

beta <= alpha

---

8. Advantages of Alpha-Beta Pruning

1. Reduces number of nodes visited
2. Speeds up search
3. Saves time
4. No change in result

---

9. Applications of Minimax

1. Chess
2. Tic-Tac-Toe
3. Checkers
4. AI Game bots
5. Strategy games
