2.4 ADVERSARIAL SEARCH

Adversarial search: minimax

•The Minimax algorithm is a fundamental strategy in adversarial search, primarily used in two-player, zero-sum games where one player's gain is the other's loss.

•It aims to determine the optimal move for a player by assuming the opponent also plays optimally.

How it works:

•Game Tree Construction:

The algorithm builds a game tree where nodes represent game states and edges represent possible moves. This tree extends to a certain depth or until terminal game states are reached.

•Maximizer and Minimizer:

The algorithm designates two players: the "Maximizer" who aims to maximize their score and the "Minimizer" who aims to minimize the Maximizer's score (which, in a zero-sum game, is equivalent to maximizing their own score).

•Evaluation Function:

At the terminal nodes (leaf nodes) of the game tree, a utility or evaluation function assigns a numerical value representing the outcome of the game from the

Maximizer's perspective (e.g., +1 for a win, -1 for a loss, 0 for a draw).

•Backtracking and Value Propagation:

The algorithm then propagates these values upwards through the tree using a recursive process:

•Min nodes: At a Minimizer's turn, the node's value is the minimum of the values of its child nodes, representing the Minimizer choosing the move that leads to the worst outcome for the Maximizer.

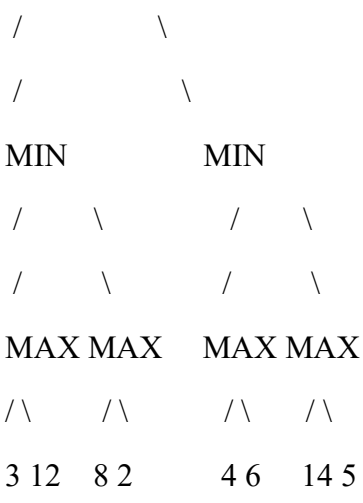
•Max nodes: At a Maximizer's turn, the node's value is the maximum of the values of its child nodes, representing the Maximizer choosing the move that leads to the best outcome for themselves.

Optimal Move Selection:

The process continues until the root node is reached. The move leading to the child node with the optimal value at the root level is the optimal move for the current player.

Minimax Algorithm Example Diagram:

Consider a simplified game tree for a two-player game, where MAX aims to maximize the score and MIN aims to minimize it. The leaf nodes represent terminal states with assigned utility values.



Evaluation Process:

•Terminal Nodes: The values at the leaf nodes (3, 12, 8, 2, 4, 6, 14, 5) are the utility values from MAX's perspective.

•MIN Nodes (Level 2):

o Left MIN node: min(3, 12) = 3 o

Middle MIN node: min(8, 2) = 2 o

Right MIN node: min(4, 6) = 4 o Far

Right MIN node: min(14, 5) = 5

- MAX Nodes (Level 1): o

Left MAX node: max(3, 2) = 3

o Right MAX node: max(4, 5) = 5

- MAX Node (Root): max(3, 5) = 5

The optimal move for MAX from the root would be to choose the branch leading to the right MAX node, as it yields the maximum possible score (5) assuming optimal play from MIN.

Simple python program

```
# Define the tree as a nested list
# Each node: [left_subtree, right_subtree] or a leaf value
tree = [
    [   # Left MIN
        [3, 12],  # Left MAX
        [8, 2]    # Right MAX
    ],
    [   # Right MIN
        [4, 6],   # Left MAX
        [14, 5]   # Right MAX
    ]
]
def minimax(node, depth, is_max):
    # If leaf node (int), return its value
    if isinstance(node, int):
        return node
    if is_max:
        # MAX node: choose max value of children
        return max(minimax(node[0], depth+1, False),
                   minimax(node[1], depth+1, False))
```

```
    else:
        # MIN node: choose min value of children
        return min(minimax(node[0], depth+1, True),
                   minimax(node[1], depth+1, True))
# Compute minimax value starting at root (MAX)
result = minimax(tree, 0, True)
print("Optimal value for MAX (Root):", result)
```

Output:

Optimal value for MAX (Root): 12

Constraint Satisfaction Problems (CSP) Definition:

1) Constraint Satisfaction Problems (CSPs) are a framework for representing and solving problems by defining variables, their possible values (domains), and the constraints that must be satisfied by the assignments to these variables.

2) The goal is to find an assignment of values to all variables from their respective domains such that all constraints are simultaneously satisfied.

Key components of a CSP:

Variables: The unknown entities in the problem that need to be assigned values.

Domains: For each variable, a set of possible values it can take. These domains can be finite and discrete, or continuous.

Constraints: Rules or conditions that restrict the combinations of values that can be assigned to the variables. These ensure that the solution adheres to the problem's requirements.

Examples of CSPs:

Sudoku:

Variables are the empty cells, domains are the numbers 1-9, and constraints ensure unique numbers in rows, columns, and 3x3 blocks.

N-Queens Problem:

Variables are the positions of N queens on an NxN chessboard, domains are the squares, and constraints ensure no two queens attack each other.

Map Coloring:

Variables are regions on a map, domains are available colors, and constraints ensure adjacent regions have different colors.

Scheduling and Timetabling:

Variables represent tasks or resources, domains are time slots or available resources, and constraints specify dependencies, deadlines, and resource availability.

Solving CSPs:

CSPs are typically solved using search algorithms, often enhanced with techniques to reduce the search space:

Backtracking Search:

A recursive algorithm that incrementally builds a solution by assigning values to variables one at a time. If an assignment violates a constraint, it backtracks and tries a different value.

Heuristics:

Strategies like Minimum Remaining Values (MRV) or Degree Heuristic can guide the order of variable selection during search to improve efficiency.

Constraint Propagation (Consistency Techniques):

Algorithms like Arc Consistency (e.g., AC3) reduce the domains of variables by removing values that cannot possibly be part of a valid solution, thereby pruning the search space before or during backtracking.

Local Search:

Algorithms that start with an initial (possibly incomplete or inconsistent) assignment and iteratively improve it by making small changes until a solution or a satisfactory state is reached.

Advantages of CSPs

Structured Problem Representation:

CSPs allow complex problems to be broken down into a structured format of variables, domains (possible values for variables), and constraints (rules to be satisfied), making the problem easier to understand and manage.

Automated Solution Finding:

Many CSP algorithms can automatically find solutions, saving time and reducing the potential for human error in complex decision-making processes.

Flexibility in Handling Constraints:

CSPs can manage both hard constraints (must be met) and soft constraints (preferences), offering a flexible approach to problems with varying levels of strictness in their rules.

Versatile Applicability:

CSPs are effective for both small and large problems, and their standardized formulation provides a common basis for solving problems across many different domains, from AI to operations research.

Enhanced Problem Understanding:

By formulating a problem as a CSP, you gain a better understanding of the problem's structure and complexity, which aids in identifying bottlenecks and potential solution paths.

Disadvantages of CSPs

High Computational Complexity:

Many CSPs, especially complex ones, can be computationally demanding, requiring significant processing power and time to find a solution.

Scalability Concerns:

While effective for many problems, the computational burden can become a significant issue when dealing with very large-scale CSPs, potentially making solutions impractical.

Requirement for Specialized Algorithms:

Solving complex CSPs often necessitates the use of specialized algorithms and combinatorial search methods, which can be intricate to develop and implement.

Difficulty with Highly Dynamic Rules:

If the rules or constraints of the problem change frequently, the CSP might need to be redefined or solved from scratch, which can be inefficient.

Applications of CSPs:

Scheduling (e.g., class timetables)

Map coloring

Puzzles (e.g., Sudoku, crosswords)

Resource allocation

Robot motion planning

AI game playing

Simple Map Coloring Program (Pure Python) Problem:

Color 3 regions (A, B, C) using 3 colors — Red, Green, Blue —

Constraint: No two adjacent regions can have the same color.

A is adjacent to B

B is adjacent to C

Python program

```
# Define regions and colors
regions = ['A', 'B', 'C']
colors = ['Red', 'Green', 'Blue']

# Adjacency constraints
adjacency = {
    'A': ['B'],
    'B': ['A', 'C'],
    'C': ['B']
}
def is_safe(region, color, assignment):
    # Check if the color can be assigned to the region
    for neighbor in adjacency.get(region, []):
        if neighbor in assignment and assignment[neighbor] == color:
```

```
            return False
    return True
def color_map(assignment={}):
    # If all regions are assigned, return assignment
    if len(assignment) == len(regions):
        return assignment
    # Select the next region to color
    for region in regions:
        if region not in assignment:
            for color in colors:
                if is_safe(region, color, assignment):
                    assignment[region] = color
                    result = color_map(assignment)
                    if result:
                        return result
                    del assignment[region]  # Backtrack
            return None  # No color fits
    return None
# Run map coloring
solution = color_map()
print("Map Coloring Solution:", solution)
```

Output:

Map Coloring Solution: {‘A’: ‘Red’, ‘B’: ‘Green’, ‘C’: ‘Red’}

Alpha Beta Pruning:

Definition:

1)Alpha-beta pruning is an optimization technique for the minimax algorithm used in game AI

2)It speeds up the search process by intelligently eliminating branches of the game tree that cannot influence the final decision

3)It does this by tracking the best maximizing and minimizing player scores using alpha and beta values, respectively, and pruning branches where a better move is already known to exist.

Key Applications:

Game AI:

This is the most prominent application. Alpha-beta pruning is extensively used in developing AI for various two-player combinatorial games, including:

Chess: Enables computers to analyze a vast number of moves and counter-moves to determine optimal strategies.

Checkers: Similar to chess, it helps the AI to evaluate game states and make intelligent moves.

Tic-Tac-Toe: While simpler, it demonstrates the pruning principle effectively in a smaller game tree.

Connect Four: Used to determine the best column to drop a piece for a winning strategy.
Gomoku: Helps the AI to search for winning patterns and block opponent's threats.

Decision-Making Algorithms:

Beyond traditional games, alpha-beta pruning can be applied in any scenario that can be modeled as a two-player adversarial search problem, where one entity tries to maximize its outcome while another tries to minimize it.

Optimization in Search Problems:

In general, it can be used in any search problem where the search space can be represented as a tree and where certain branches can be identified as irrelevant to the final optimal solution early in the search.

Advantages

Reduced Complexity

It avoids exploring large parts of the game tree that cannot influence the final decision, significantly reducing the number of nodes evaluated compared to the standard Minimax algorithm.

Faster Computation:

By pruning unnecessary branches, the algorithm makes decisions more quickly.

Deeper Search:

The reduced computational burden allows the algorithm to explore the game tree to a greater depth within the same time frame, leading to more strategic and betterinformed choices.

Optimal Decisions:

Despite cutting off branches, alpha-beta pruning still guarantees the same optimal move as the Minimax algorithm.

Disadvantages

Order-Dependent Effectiveness:

The amount of pruning achieved is highly dependent on the order in which moves are explored; it performs best when the best possible moves are evaluated first.

Requires Extra Logic:

Implementing alpha-beta pruning requires additional code and logic to track the alpha and beta values throughout the search.

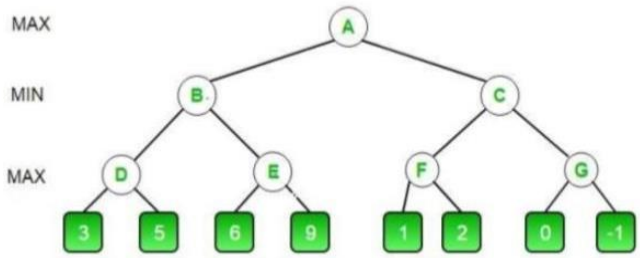
Limited Applicability:

It is primarily used for deterministic, two-player games with perfect information (like chess) and is not suitable for probabilistic or real-time decision-making scenarios.

Not a Universal Solution:

It doesn’t eliminate all the problems associated with the base Minimax algorithm, and for poorly ordered game trees, the gains in performance can be minimal.

Example diagram with explanation



The initial call starts from A. The value of alpha here is -INFINITY and the value of beta is +INFINITY. These values are passed down to subsequent nodes in the tree. At A the maximizer must choose max of B and C, so A calls B first

At B it the minimizer must choose min of D and E and hence calls D first.

At D, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at D is max(-INF, 3) which is 3.

To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.

D now looks at its right child which returns a value of 5.At D, alpha = max(3, 5) which is

5. Now the value of node D is 5

D returns a value of 5 to B. At B, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. B now calls E to see if he can get a lower value than 5.

At E the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at B and that is what B passed down to E

Now E looks at its left child which is 6. At E, alpha = max(-INF, 6) which is 6. Here the condition becomes true. Beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and E returns 6 to B

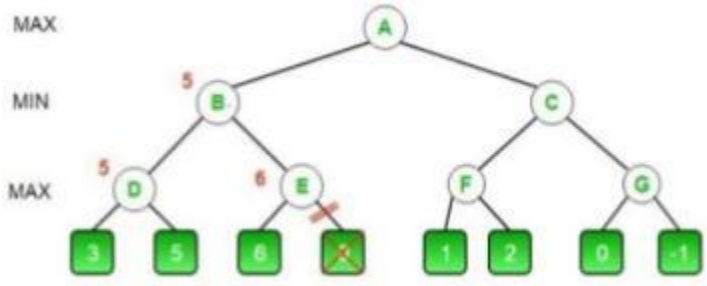
Note how it did not matter what the value of E's right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never

come this way because he can get a 5 on the left side of B. This way we didn't have to look at that 9 and hence saved computation time.

E returns a value of 6 to B. At B, beta = min(5, 6) which is 5.The value of node B is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed.

Alpha Beta Pruning 2



B returns 5 to A. At A, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. A now calls C to see if it can get a higher value than 5.

At C, alpha = 5 and beta = +INF. C calls F

At F, alpha = 5 and beta = +INF. F looks at its left child which is a 1. Alpha = max(5, 1) which is still 5.

F looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5

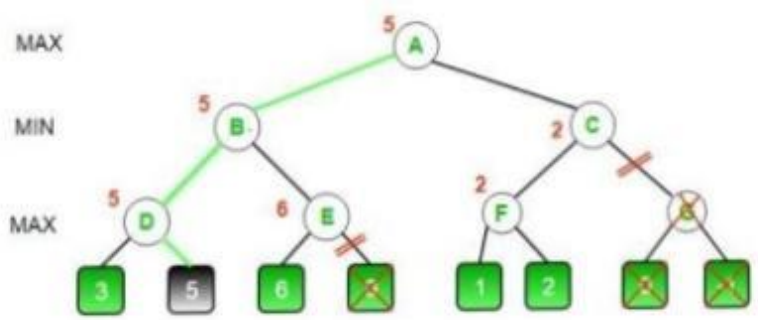
F returns a value of 2 to C. At C, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of G.

The intuition behind this break-off is that, at C the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose B. So why would the maximizer ever choose C and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub-tree.

C now returns a value of 2 to A. Therefore the best value at A is max(5, 2) which is a 5.

Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see G has been crossed out as it was never computed.



Program:

```
def minimax():
    alpha = float('-inf')  # Best value for MAX so far
    beta = float('inf')    # Best value for MIN so far

    # First MIN branch
    A = min(3, 5)  # MIN chooses 3
    alpha = max(alpha, A)  # Update alpha
    print(f"After first MIN branch: A = {A}, alpha = {alpha}")

    # Second MIN branch
    B = min(6, 9)  # MIN chooses 6
    if B <= alpha:
        print("Pruned! (B <= alpha)")
    else:
        alpha = max(alpha, B)
        print(f"After second MIN branch: B = {B}, alpha = {alpha}")
```

```
    print("Best value for MAX:", alpha)
# Run the function
minimax()
```

Explanation

MAX has 2 options:

MIN(3, 5) → 3

MIN(6, 9) → 6

After getting 3, MAX sees no need to check 6 if it's worse for MAX (if $6 \leq 3$) → Prune!