UNIT-II: Searching and Game Playing

Contents

---

1. Searching for Solutions

Definition and Importance of Search

Searching for solutions is a fundamental technique in artificial intelligence used to find a sequence of actions that leads from an initial state to a goal state[1]. According to Mahesh Huddar's lectures, the search process involves exploring the problem space systematically to discover the optimal or satisfactory path[1].

Key Components of Search:

- **Initial State:** The starting point of the search
- **Goal State:** The desired outcome to be achieved
- **Actions:** Possible moves or transitions available at each state
- **Path Cost:** The cost associated with each action
- **Search Space:** The complete set of all possible states that can be reached[1]

Why Search is Important

Search algorithms are essential in AI for:

- Finding optimal routes in navigation systems[1]
- Solving puzzles and constraint satisfaction problems[1]
- Game playing and strategic decision-making[1]
- Planning and scheduling applications[1]

---

2. Uninformed (Blind) Search Strategies

Uninformed search algorithms, also known as blind search algorithms, do not use any domain-specific knowledge or heuristics about the problem being solved[2]. They only rely on the problem definition itself and explore the search space systematically[2].

2.1 Breadth-First Search (BFS)

Definition:

Breadth-First Search is a searching algorithm that explores all nodes at the current depth level before moving to nodes at the next depth level[2]. It uses a queue data structure (FIFO - First In First Out) to manage the frontier of nodes to be explored[2].

Algorithm Steps (Mahesh Huddar):

1. Create a node list (queue) and add the initial state
2. If the queue is empty, return failure
3. Remove the first node from the queue
4. If this node is the goal state, return success
5. Otherwise, generate all child nodes and add them to the end of the queue
6. Repeat from step 2[3]

Key Characteristics:

- **Complete:** BFS guarantees finding a solution if one exists[2]
- **Optimal:** Finds the shortest path in an unweighted graph[2]
- **Time Complexity:** O(b^d), where b is the branching factor and d is the depth
- **Space Complexity:** O(b^d) - significant memory requirement[2]

Advantages:

- Finds the shortest path between starting point and goal
- Always finds optimal solutions
- Systematic exploration ensures no paths are missed
- Finds the closest goal in less time[2]

Disadvantages:

- Very high memory usage for large search spaces
- Slow for problems where the goal is deep in the tree
- Cannot find solutions beyond the current depth until all nodes at that depth are explored[2]

YouTube Reference:

Mahesh Huddar - "Breadth-First Search Algorithm Solved Example"
URL: https://www.youtube.com/watch?v=6B0mSyyFE2M

GeeksforGeeks Reference:

Breadth First Search (BFS) for Artificial Intelligence
URL: https://www.geeksforgeeks.org/artificial-intelligence/breadth-first-search-bfs-for-artificial-intelligence/

2.2 Depth-First Search (DFS)

Definition:

Depth-First Search is a searching algorithm that explores as far as possible along each branch before backtracking[2]. It uses a stack data structure (LIFO - Last In First Out) to manage the frontier of nodes to be explored[2].

Algorithm Steps (Mahesh Huddar):

1. Create a node stack and add the initial state
2. If the stack is empty, return failure
3. Remove the top node from the stack
4. If this node is the goal state, return success
5. Otherwise, generate all child nodes and add them to the top of the stack (in reverse order)
6. Repeat from step 2[3]

Key Characteristics:

- **Incomplete:** May not find a solution if the search space is infinite or contains cycles
- **Non-Optimal:** Does not guarantee finding the shortest path
- **Time Complexity:** O(b^m), where m is the maximum depth
- **Space Complexity:** O(bm) - much better memory usage than BFS[2]

Advantages:

- Uses less memory compared to BFS as it stores only a single path at a time
- May find solutions without examining much of the search space
- Efficient for deep solution paths
- Works well in many practical applications[3]

Disadvantages:

- Can get trapped in infinite loops if cycles exist
- Does not guarantee shortest path
- May explore deeply before finding the goal
- Not suitable for deep or infinite search spaces without modifications[3]

YouTube Reference:

Mahesh Huddar - "Depth First Search Algorithm Solved Example Advantages and Disadvantages"

URL: https://www.youtube.com/watch?v=cq0cBynFFXI

2.3 Depth-Limited Search (DLS)

Definition:

Depth-Limited Search is a variation of Depth-First Search that imposes a maximum depth limit on the search[2]. It prevents DFS from exploring infinitely deep and returns failure if the goal is not found within the specified depth limit[2].

Key Characteristics:

- Prevents infinite loops by setting a depth threshold
- Still uses less memory than BFS
- Cannot find solutions beyond the depth limit
- Time Complexity: O(b^l), where l is the depth limit
- Space Complexity: O(bl)[2]

Advantages:

- Prevents infinite exploration in deep search spaces
- Memory efficient compared to BFS
- Useful when the solution depth is known or estimated[2]

Disadvantages:

- Will fail if the goal is deeper than the depth limit
- Requires prior knowledge of appropriate depth limit
- May still be inefficient for problems with unknown solution depth[2]

YouTube Reference:

Mahesh Huddar - "Solved Example Depth Limited Depth First Search (DLDFS)"
URL: https://www.youtube.com/watch?v=P7WQUBLKDmo

2.4 Iterative Deepening Depth-First Search (IDDFS/IDS)

Definition:

Iterative Deepening Depth-First Search combines the benefits of BFS and DFS by running DFS with increasing depth limits until a solution is found[2]. It performs DFS with depth limit 0, then depth limit 1, then depth limit 2, and so on until the goal is reached[2].

Algorithm Steps:

1. Set depth_limit = 0
2. Perform DFS with current depth_limit
3. If goal is found, return success
4. If goal is not found, increment depth_limit by 1
5. Repeat until goal is found or maximum depth is reached[2]

Key Characteristics:

- **Complete:** Guarantees finding a solution if one exists

- **Optimal:** Finds the shortest path like BFS
- **Time Complexity:** O(b^d) - same as BFS but with larger constant
- **Space Complexity:** O(bd) - much better than BFS
- Combines optimality of BFS with space efficiency of DFS[2]

Advantages:

- Ensures completeness and optimality
- Uses significantly less memory than BFS
- Effective for problems where solution depth is unknown
- Combines the best properties of BFS and DFS[2]

Disadvantages:

- Repeats exploration at shallower depths multiple times
- Time complexity similar to BFS in worst case
- May be slower than BFS in practice due to redundant exploration[2]

YouTube Reference:

Mahesh Huddar - "IDS Search | DFS Algorithm in Artificial Intelligence"
URL: https://www.youtube.com/watch?v=BK8cEWKHCkY

2.5 Uniform Cost Search (UCS)

Definition:

Uniform Cost Search extends BFS by considering the cost of each move. It always expands the node with the lowest cumulative path cost from the start node[2]. UCS uses a priority queue to manage nodes based on their total cost from the initial state[2].

Algorithm Steps:

1. Create a priority queue and add the initial state with cost 0
2. If the priority queue is empty, return failure
3. Remove the node with minimum cumulative cost
4. If this node is the goal state, return success
5. Otherwise, generate all child nodes, calculate their costs, and add to the priority queue
6. Repeat from step 2[2]

Key Characteristics:

- **Complete:** Guarantees finding a solution if one exists
- **Optimal:** Finds the minimum-cost path when all costs are non-negative
- **Time Complexity:** O(b^(1+⌊C*/ε⌋)), where C* is the optimal cost and ε is minimum cost
- **Space Complexity:** O(b^(1+⌊C*/ε⌋))[2]

Advantages:

- Finds the least-cost path, not just shortest path
- Optimal solution when all costs are non-negative
- Effective for weighted graphs where different actions have different costs[2]

Disadvantages:

- Slower than BFS in uniform cost cases
- Higher memory requirements with large branching factors
- Requires complete exploration to guarantee optimal solution[2]

GeeksforGeeks Reference:

Uniform Cost Search (UCS) in AI
URL: https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/

---

3. Heuristic (Informed) Search Strategies

3.1 Definition and Importance of Heuristics

What is a Heuristic?

A heuristic is an informed guess or estimate of how close a given state is to the goal state[4]. It provides domain-specific knowledge to guide the search toward the goal more efficiently than blind search strategies[4].

Heuristic Function h(n):

The heuristic function h(n) estimates the cost of the cheapest path from node n to the goal node[4]. Different problems require different heuristic functions suited to their specific characteristics[4].

Why Heuristics Matter:

- Significantly reduce the search space compared to uninformed search
- Guide the search toward promising areas of the problem space
- Allow finding solutions faster with less memory[4]
- Improve overall efficiency of search algorithms

Properties of a Good Heuristic:

- **Admissibility:** The heuristic never overestimates the actual cost to reach the goal ($h(n) \leq$ actual cost)[4]
- **Consistency (Monotonicity):** The heuristic estimate should satisfy the triangle inequality - the cost estimate does not decrease along a path[4]
- **Informativeness:** A more informative heuristic (closer to actual cost) reduces search space exploration[4]

YouTube Reference:

Sudhakar Atchala - "Heuristic Search and Heuristic Function in AI"
URL: https://www.youtube.com/watch?v=WCTvVpKx790

GeeksforGeeks Reference:

Heuristic Search Techniques in AI
URL: https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/

3.2 Hill Climbing

Definition:

Hill Climbing is a local search algorithm that continuously moves in the direction of increasing value to find the peak (best solution) of the state space landscape[4]. It is a simple heuristic search algorithm that evaluates neighbors of the current state and moves to the neighbor with the best value[4].

Algorithm Steps (Mahesh Huddar):

1. Evaluate the initial state
2. If it is the goal state, return it and quit; otherwise, continue with it as current state
3. For each successor of the current state:
    - Evaluate the successor
    - If the successor has better value than current state, make it the new current state
4. If no successor improves the current state, return current state (local maximum reached)
5. Otherwise, repeat from step 3[4]

Types of Hill Climbing:

1. Simple Hill Climbing:

- Moves to the first neighbor that is better than the current state
- Fast but may miss better solutions
- YouTube Reference: https://www.youtube.com/watch?v=deT2kSJQKEI

2. Steepest Ascent Hill Climbing (Gradient Search):

- Evaluates all neighbors and moves to the neighbor with the best value
- More thorough but computationally expensive
- YouTube Reference: https://www.youtube.com/watch?v=ppV0sJUfqhQ

Key Characteristics:

- **Local Search:** Operates using only local information
- **Greedy:** Always chooses the locally best option
- **Incomplete:** May not find a solution even if one exists
- **Non-Optimal:** May reach local maxima instead of global maximum[4]

Problems with Hill Climbing:

- **Local Maximum:** Gets stuck at a local peak that is not the global optimum
- **Flat Maximum (Plateau):** Reaches a region where all neighbors have similar values
- **Ridges:** Special form of local maximum with multiple peaks where single moves cannot reach better solutions[4]

Solutions to Hill Climbing Problems:

- **Backtracking:** When stuck at local maximum, select different initial state and restart
- **Bidirectional Search:** Search from both start and goal to overcome ridge problems

- **Random Restart:** Randomly restart from new initial states and keep best result found[4]

Advantages:

- Simple and easy to implement
- Memory efficient as it only keeps track of current state
- Fast for many practical problems with good heuristics[4]

Disadvantages:

- Gets stuck at local maxima
- Cannot guarantee finding global optimum
- Sensitive to initial state and heuristic quality
- May fail in multimodal landscapes[4]

YouTube Reference:

Mahesh Huddar - "Hill climbing Search Algorithm-Artificial Intelligence-UNIT 2"
URL: https://www.youtube.com/watch?v=3OO3vWTpFqY

3.3 Best-First Search (Greedy Search)

Definition:

Best-First Search is an informed search algorithm that always expands the node that appears closest to the goal according to the heuristic function[4]. It uses a priority queue ordered by the heuristic value h(n)[4].

Algorithm Steps:

1. Add the initial state to the priority queue
2. While the priority queue is not empty:
    - Remove the node with the best (lowest) h(n) value
    - If this is the goal, return success
    - Otherwise, expand the node and add its children to the priority queue
3. If queue becomes empty, return failure[4]

Key Characteristics:

- **Informed:** Uses heuristic function to guide search
- **Greedy:** Always selects the node closest to goal according to heuristic
- **Incomplete:** May not find a solution if heuristic misleads search
- **Non-Optimal:** May not find the optimal (lowest cost) path[4]

Advantages:

- Generally faster than uninformed search
- Uses domain knowledge to guide search
- Better memory efficiency than uninformed searches in many cases[4]

Disadvantages:

- Not guaranteed to find optimal solution

- Can fail if heuristic is misleading
- May explore many nodes before finding goal[4]

YouTube Reference:

Mahesh Huddar - "Best First Search (BFS) Algorithm"
URL: https://www.youtube.com/watch?v=Gbw1IsnY7KE

3.4 A* Algorithm

Definition:

A* Algorithm is one of the most popular and effective heuristic search algorithms. It combines the benefits of Dijkstra's algorithm and a heuristic function to find the optimal path from a start node to a goal node[5]. It uses the evaluation function f(n) = g(n) + h(n) to prioritize which nodes to explore[5].

*Key Components of A:**

1. g(n) - Actual Cost:

- The accumulated cost of the path from the start node to node n[5]
- Represents the known cost already incurred
- Updated as nodes are explored

2. h(n) - Heuristic Cost:

- An estimate of the cost to reach the goal from node n[5]
- Domain-specific estimate (never overestimates the actual cost if admissible)
- Guides the search toward the goal

3. f(n) - Total Estimated Cost:

- f(n) = g(n) + h(n)
- The total estimated cost of the cheapest solution through node n[5]
- Used to order nodes in the priority queue

Algorithm Steps:

1. Initialize: Add the initial node to the open set with f(n) = h(n)
2. While open set is not empty:
    - Select node with lowest f(n) value from open set
    - If this is the goal, return the path
    - Otherwise, expand the node and evaluate all neighbors
    - For each neighbor: Calculate g(n), h(n), and f(n)
    - If better path found to neighbor, update it and add to open set
3. If open set becomes empty, return failure (no path exists)[5]

Key Characteristics:

- **Complete:** Guarantees finding a solution if one exists
- **Optimal:** Finds the lowest-cost path with an admissible heuristic[5]
- **Efficient:** Explores fewer nodes than uninformed algorithms with good heuristics
- **Time Complexity:** Depends on heuristic quality; O(b^d) worst case

- **Space Complexity:** Stores all nodes in memory - O(b^d)[5]

Common Heuristic Functions:

- **Manhattan Distance:** Sum of absolute differences in x and y coordinates
  - Used for grid-based pathfinding
  - $h(n) = |x_1 - x_2| + |y_1 - y_2|$[5]
- **Euclidean Distance:** Straight-line distance between two points
  - Used for continuous spaces
  - $h(n) = \sqrt{[(x_1 - x_2)^2 + (y_1 - y_2)^2]}$[5]
- **Chebyshev Distance:** Maximum difference in coordinates
  - Used when diagonal movement is allowed
  - $h(n) = \max(|x_1 - x_2|, |y_1 - y_2|)$[5]

Advantages:

- Guarantees optimal solution with admissible heuristic
- Much faster than uninformed search with good heuristics
- Explores fewer nodes than other algorithms
- Flexible and adaptable to different problem domains[5]

Disadvantages:

- High memory usage for large search spaces
- Efficiency depends heavily on heuristic quality
- Poor heuristic can make it slower than uninformed search
- Computational overhead in complex graphs[5]

YouTube References:

Dr. Mahesh Huddar discusses A* in various lectures on informed search algorithms.

GeeksforGeeks Reference:

A* algorithm and its Heuristic Search Strategy in AI
URL: https://www.geeksforgeeks.org/artificial-intelligence/a-algorithm-and-its-heuristic-search-strategy-in-artificial-intelligence/

3.5 AO* Algorithm (A-OR* Algorithm)

Definition:

AO* (A-OR*) Algorithm is an informed search algorithm used for solving problems that can be decomposed into subproblems[6]. It uses AND/OR graphs to represent problem decomposition and is based on problem reduction strategy[6].

Problem Reduction Approach:

Rather than searching through states, AO* decomposes complex problems into simpler subproblems using AND/OR graphs:

- **AND Nodes:** Represent problem decomposition where ALL subproblems must be solved[6]

- **OR Nodes:** Represent alternative solutions where ANY ONE subproblem can lead to solution[6]
- Problem is reduced through multiple levels until primitive (terminal) goals are reached[6]

Key Concepts:

- **Search Graph:** Dynamically constructed during search (not static game tree like minimax)
- **Solution Graph:** A subgraph that represents the solution path
- **Cost Function:** f(n) = g(n) + h(n), same as A*[6]

Algorithm Steps:

1. Start with the root node (complex problem)
2. Evaluate the heuristic value f(n) of current node
3. Find the most promising node with minimum f(n) value
4. Expand the most promising node:
    - If AND node, expand all successors
    - If OR node, expand all successors and select best path
5. Update the cost of each path after finding terminal nodes
6. Mark solved nodes (terminal nodes with cost 0)
7. Propagate solved status up the graph
8. Continue until the root is marked as solved[6]

Key Characteristics:

- **AND/OR Graphs:** Represents both problem composition and alternatives
- **Problem Reduction:** Divides problems into manageable subproblems
- **Optimal:** Finds optimal solution with admissible heuristic
- **Efficient:** Explores only relevant solution paths[6]

Advantages:

- Naturally handles hierarchical problem decomposition
- More efficient for problems with clear subproblem structure
- Can prune entire branches representing unsolvable subproblems
- Optimal path guaranteed with good heuristic[6]

Disadvantages:

- More complex to implement than A*
- Requires problem to be decomposable into subproblems
- Less widely applicable than A* for general pathfinding[6]

YouTube Reference:

Mahesh Huddar - "AO* algorithm in AI (artificial intelligence) || AND OR Graph"
URL: https://www.youtube.com/watch?v=9bjsOhSlwVM

GeeksforGeeks Reference:

Difference Between A* and AO* Algorithm
URL: https://www.geeksforgeeks.org/artificial-intelligence/difference-between-a-and-ao-alo

githm/

---

4. Problem Reduction

Definition

Problem Reduction is a technique where a complex problem is decomposed into a set of simpler subproblems that are solved independently[6]. Once all subproblems are solved, their solutions are combined to solve the original problem[6].

Relationship with AND/OR Graphs

Problem reduction is represented using AND/OR graphs:

- **OR Links:** Indicate alternative approaches to solve a problem (choose one)
- **AND Links:** Indicate that multiple subproblems must be solved (solve all)[6]

Example: Proving a Mathematical Theorem

To prove a theorem, you might reduce it into:

- Lemma 1 AND Lemma 2 AND Lemma 3 (all must be proven)
- OR alternative direct proof (only one approach needed)[6]

Applications

- Theorem proving in logic
- Game playing with complex move spaces
- Robotics path planning with hierarchical tasks
- Software architecture and system design[6]

---

5. Game Playing - Adversarial Search

5.1 Games and Game Theory

Definition of Games in AI:

In artificial intelligence, a game is a formal problem where:

- There are two or more players (usually two in adversarial games)
- Each player has a set of possible actions/moves
- The game has alternating turns between players
- There is a terminal state where the game ends with a clear winner, loser, or draw
- Each terminal state has a utility value (payoff) for each player[7]

Game Types:

- **Perfect Information Games:** Both players know the complete state of the game (chess, tic-tac-toe)
- **Imperfect Information Games:** Players have incomplete information (poker, bridge)
- **Zero-Sum Games:** One player's gain is another's loss
- **Non-Zero-Sum Games:** Both players can gain or lose[7]

Game Tree Representation:

Games are represented as trees where:

- Root node represents the initial game state
- Each node represents a game state
- Each edge represents a possible move
- Leaf nodes represent terminal states with utility values[7]

Problems in Game Playing:

- **Computational Complexity:** Game trees grow exponentially with depth
- **Search Space Explosion:** Chess has ~10^120 possible positions
- **Evaluation Function Quality:** Depends on ability to estimate position strength
- **Time Constraints:** Must make decisions within time limits[7]
- **Incomplete Information:** Many games don't reveal all information to players[7]

5.2 Minimax Algorithm

Definition:

The Minimax algorithm is a decision-making algorithm used in game theory and computer games to find the optimal move for a player, assuming both players play optimally[7]. It alternates between maximizing and minimizing players, exploring the game tree to find the best strategy[7].

Key Concepts:

Maximizing Player (Max):

- Tries to maximize the utility value (score)
- Chooses the move with the highest utility
- Represented as square (□) nodes in game trees[7]

Minimizing Player (Min):

- Tries to minimize the maximizer's utility value
- Chooses the move with the lowest utility for opponent
- Represented as circle (○) nodes in game trees[7]

Assumption of Optimal Play:
Both players are assumed to play optimally - each player makes the best possible move given the current state[7]

Algorithm Steps:

1. Generate the complete game tree from current state to terminal states
2. Assign utility values to all terminal states:
    - +1 if Max player wins
    - 0 if draw
    - -1 if Min player wins
3. Backup utility values from terminal nodes to root using minimax formula:
    - At Max nodes: Select the maximum value from children
    - At Min nodes: Select the minimum value from children

4. At the root, Max player selects the move leading to the child with maximum value[7]

Minimax Formulas:

For Maximizing Player:

MINIMAX(s) = max_{a ∈ A(s)} MINIMAX(RESULT(s, a))

For Minimizing Player:

MINIMAX(s) = min_{a ∈ A(s)} MINIMAX(RESULT(s, a))

For Terminal States:

MINIMAX(s) = UTILITY(s)[7]

Key Characteristics:

- **Complete:** Always finds a solution
- **Optimal:** Guarantees optimal move assuming opponent plays optimally[7]
- **Exhaustive:** Explores the entire game tree
- **Time Complexity:** O(b^m), where b is branching factor and m is depth
- **Space Complexity:** O(bm) for depth-first search implementation[7]

Advantages:

- Mathematically sound approach to game playing
- Guarantees best move assuming perfect play
- Simple conceptual framework
- Optimal decision making[7]

Disadvantages:

- Computationally expensive for large game trees
- Exponential time complexity makes it impractical for deep games
- Must explore entire game tree to reach terminal states
- Not practical for real-time game playing without optimizations[7]

Example: Tic-Tac-Toe Game Tree

In tic-tac-toe with minimax:

- Root represents empty board (Max's turn to move first)
- Each level alternates between Max and Min moves
- Leaf nodes show outcomes: +1 (Max wins), 0 (draw), -1 (Min wins)
- Minimax propagates values upward to find best move[7]

YouTube References:

Mahesh Huddar - "MiniMax Search Artificial Intelligence Solved Example"
URL: https://www.youtube.com/watch?v=tDv7lrklaQE

Mahesh Huddar - "MiniMax Search Algorithm Solved Example"
URL: https://www.youtube.com/watch?v=MzmHcqPFd18

Mahesh Huddar - "Tic Tac Toe Game Playing using Minimax"
URL: https://www.youtube.com/watch?v=_uFVLU3RTxQ

GeeksforGeeks Reference:

Mini-Max Algorithm in Artificial Intelligence
URL: https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/

5.3 Optimal Decisions in Multiplayer Games

Extending Minimax to Multiplayer Scenarios:

While basic minimax handles two-player games, multiplayer games require modifications:

- Instead of max and min values, use a value vector for each player
- Each node returns a vector of utility values, one per player
- Each player selects moves that maximize their own value[7]

Decision Making Strategy:

1. For each possible move, calculate the resulting game state
2. Recursively evaluate each resulting state assuming optimal play by all players
3. Each player selects the move that maximizes their utility value
4. Propagate these values back through the game tree[7]

Challenges in Multiplayer Games:

- **Coalition Formation:** Players may form alliances that complicate analysis
- **Preference Modeling:** Must accurately represent each player's utility function
- **Computational Complexity:** Multiplies with number of players
- **Cooperation vs. Competition:** Balance between collaborative and competitive moves[7]

5.4 Alpha-Beta Pruning

Definition:

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm that significantly reduces the number of nodes evaluated in the game tree by pruning (eliminating) branches that cannot affect the final decision[8].

Key Insight:

Alpha-Beta pruning recognizes that once a branch is determined to be worse than a previously explored alternative, there is no need to explore it further[8].

Alpha (α) and Beta (β) Parameters:

Alpha (α) - Maximum Lower Bound:

- The best (highest) value that the Maximizing player is assured of[8]
- Represents what Max can guarantee at minimum
- Initially set to -∞

Beta (β) - Minimum Upper Bound:

- The best (lowest) value that the Minimizing player is assured of[8]
- Represents what Min can guarantee at maximum
- Initially set to +∞

Pruning Conditions:

Alpha Cutoff (at Min nodes):

- When $\beta \leq \alpha$, prune the remaining successors of the Min node
- The Max player has found a better move elsewhere[8]

Beta Cutoff (at Max nodes):

- When $\alpha \geq \beta$, prune the remaining successors of the Max node
- The Min player will prevent Max from reaching this node[8]

Algorithm Steps:

1. Start with $\alpha = -\infty$ and $\beta = +\infty$ at the root
2. At Max nodes:
    - For each successor: recursively evaluate with current α and β
    - Update $\alpha = \max(\alpha, \text{child\_value})$
    - If $\alpha \geq \beta$, prune remaining children (beta cutoff)
3. At Min nodes:
    - For each successor: recursively evaluate with current α and β
    - Update $\beta = \min(\beta, \text{child\_value})$
    - If $\beta \leq \alpha$, prune remaining children (alpha cutoff)
4. Return the best value found[8]

Key Characteristics:

- **Same Result:** Produces the same optimal move as minimax
- **Improved Efficiency:** Can reduce nodes evaluated from O(b^m) to O(b^(m/2))[8]
- **Best Case:** Can explore twice as deep in same time
- **Worst Case:** No pruning if moves ordered badly
- **Ordering Dependent:** Effectiveness depends on move order[8]

Move Ordering for Better Performance:

The effectiveness of alpha-beta pruning depends on the order in which moves are evaluated:

- **Best Case:** Evaluate moves in order of best to worst - maximum pruning occurs
- **Worst Case:** Evaluate moves in order of worst to best - no pruning
- **Killer Heuristic:** Moves that caused cutoffs in siblings are likely to cause cutoffs in other branches[8]
- **Transposition Table:** Store previously computed positions to avoid re-evaluation

Advantages:

- Dramatically reduces computation time and search space
- Maintains optimal solution guarantee

- Allows exploration of deeper game trees in real-time[8]
- Proven effective in chess engines and game AI

Disadvantages:

- Effectiveness depends heavily on move ordering
- Requires careful implementation to realize benefits
- Poor move ordering can provide no benefit[8]

Example: Alpha-Beta Pruning in Practice

In a chess engine:

- Best moves are evaluated first (based on previous analysis)
- Once a good move is found, bad branches are quickly pruned
- This allows evaluating 5-6 moves deeper than minimax alone[8]

YouTube References:

Mahesh Huddar - "1. Alpha Beta Pruning (Cutoff) Search Algorithm Solved Example"
URL: https://www.youtube.com/watch?v=9D1hVGumxCo

Sudhakar Atchala - "Alpha-Beta Pruning Search Algorithm with Solved Example"
URL: https://www.youtube.com/watch?v=un92YiweX9M

Mahesh Huddar - "Alpha Beta Pruning Algorithm in AI"
URL: https://www.youtube.com/watch?v=VWUKrMEUZKo

GeeksforGeeks Reference:

Alpha-Beta pruning in Adversarial Search Algorithms
URL: https://www.geeksforgeeks.org/artificial-intelligence/alpha-beta-pruning-in-adversarial-search-algorithms/

5.5 Evaluation Functions

Definition:

An evaluation function (or static evaluation function) is a heuristic that estimates the desirability of a game position without searching further[9]. It assigns a numerical score to non-terminal states to guide the search[9].

Purpose of Evaluation Functions:

- **Depth Limiting:** Allows terminating search before reaching terminal states in deep trees
- **Position Assessment:** Provides quick estimate of position strength
- **Move Ordering:** Helps order promising moves first for better alpha-beta pruning
- **Real-Time Decision Making:** Enables game-playing within time constraints[9]

Characteristics of Good Evaluation Functions:

- **Accuracy:** Should estimate position accurately without extensive search
- **Efficiency:** Must compute quickly in reasonable time
- **Consistency:** Should assign similar scores to similar positions

- **Domain Knowledge:** Should incorporate expert knowledge about the game
- **Sensitivity:** Should distinguish between good and bad positions[9]

Examples of Evaluation Functions:

Chess Evaluation Function:

Typically combines:

- Material value: Points for pieces (pawn=1, knight=3, bishop=3, rook=5, queen=9)
- Position value: Centralization, king safety, pawn structure
- Mobility: Number of available moves
- Control of center squares[9]

Total Score = $w_1 \times$ Material + $w_2 \times$ Position + $w_3 \times$ Mobility[9]

Tic-Tac-Toe Evaluation Function:

e(s) = (number of 3-in-a-rows for computer) - (number of 3-in-a-rows for opponent)[9]

How Evaluation Functions are Used in Search:

1. At each node during minimax search:
    - If depth limit reached, apply evaluation function
    - Return estimated score instead of searching further
2. Alpha-beta pruning uses evaluation function values to determine pruning decisions
3. Better evaluation function → better game playing performance[9]

Challenges in Designing Evaluation Functions:

- **Balance Accuracy vs. Speed:** Detailed evaluation takes more time
- **Feature Selection:** Choosing relevant position features
- **Weight Learning:** Determining appropriate weights for different features
- **Domain Complexity:** Different games require different features
- **Horizon Problem:** Cannot see threats just beyond search depth[9]

Learning Evaluation Functions:

Modern approaches use:

- **Machine Learning:** Train neural networks on game databases
- **Reinforcement Learning:** Learn from self-play experience
- **Deep Learning:** AlphaZero uses neural networks for evaluation[9]

YouTube Reference:

Deeba Kannan - "Artificial Intelligence - Min Max Algorithm"
URL: https://www.youtube.com/watch?v=J0Km3aOrDVE

Key Learning Resources

YouTube Channels and Lectures

Dr. Mahesh Huddar (HIT, Nidasoshi):

- Channel: https://www.youtube.com/@MaheshHuddar
- Key Videos for Unit-II:
  - Breadth-First Search Algorithm: https://www.youtube.com/watch?v=6B0mSyyFE2M
  - Depth First Search Algorithm: https://www.youtube.com/watch?v=cq0cBynFFXI
  - Depth Limited Search: https://www.youtube.com/watch?v=P7WQUBLKDmo
  - Iterative Deepening Search: https://www.youtube.com/watch?v=BK8cEWKHCkY
  - Hill Climbing Search: https://www.youtube.com/watch?v=3OO3vWTpFqY
  - Best First Search: https://www.youtube.com/watch?v=Gbw1IsnY7KE
  - A* Algorithm References
  - AO* Algorithm: https://www.youtube.com/watch?v=9bjsOhSlwVM
  - Minimax Algorithm: https://www.youtube.com/watch?v=tDv7lrklaQE
  - Alpha-Beta Pruning: https://www.youtube.com/watch?v=9D1hVGumxCo
  - Infrastructure for Search: https://www.youtube.com/watch?v=AlpcDB-J23M

Sudhakar Atchala:

- Heuristic Search: https://www.youtube.com/watch?v=WCTvVpKx790
- Steepest Ascent Hill Climbing: https://www.youtube.com/watch?v=ppV0sJUfqhQ
- Alpha-Beta Pruning: https://www.youtube.com/watch?v=un92YiweX9M
- AI Complete Course Playlist: https://www.youtube.com/playlist?list=PLXj4XH7LcRfBcN_qu_3OpmCYUQfctfxX5

GeeksforGeeks Comprehensive Resources

1. **Uninformed Search Algorithms:** https://www.geeksforgeeks.org/artificial-intelligence/uniformed-search-algorithms-in-ai/
2. **Breadth-First Search (BFS):** https://www.geeksforgeeks.org/artificial-intelligence/breadth-first-search-bfs-for-artificial-intelligence/
3. **Uniform Cost Search (UCS):** https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/
4. **Heuristic Search Techniques:** https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/
5. *A Algorithm:** https://www.geeksforgeeks.org/artificial-intelligence/a-algorithm-and-its-heuristic-search-strategy-in-artificial-intelligence/
6. *Difference Between A and AO*:** https://www.geeksforgeeks.org/artificial-intelligence/difference-between-a-and-ao-alogithm/
7. **Adversarial Search Algorithms:** https://www.geeksforgeeks.org/artificial-intelligence/adversarial-search-algorithms/
8. **Mini-Max Algorithm:** https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/
9. **Alpha-Beta Pruning:** https://www.geeksforgeeks.org/artificial-intelligence/alpha-beta-pruning-in-adversarial-search-algorithms/

Summary Table: Search Algorithms Comparison

Algorithm	Complete?	Optimal?	Time Complexity	Space Complexity
Breadth-First Search	Yes	Yes*	O(b^d)	O(b^d)
Depth-First Search	No	No	O(b^m)	O(bm)
Depth-Limited Search	No	No	O(b^l)	O(bl)
Iterative Deepening	Yes	Yes*	O(b^d)	O(bd)
Uniform Cost Search	Yes	Yes	O(b^(1+C*/ε))	O(b^(1+C*/ε))
Hill Climbing	No	No	O(1) per step	O(1)
Best-First Search	No	No	O(b^d)	O(b^d)
A* Algorithm	Yes	Yes**	Depends on h(n)	O(b^d)
AO* Algorithm	Yes	Yes**	Depends on h(n)	Depends on problem
Minimax	Yes	Yes***	O(b^m)	O(bm)
Alpha-Beta Pruning	Yes	Yes***	O(b^(m/2))	O(bm)

Table 1: Comparison of Search Algorithms (* in uniform cost trees, ** with admissible heuristic, *** in two-player games)

References

[1] Huddar, M. (2022). Tasks in AI | Steps to Solve Problem. Retrieved from https://www.youtube.com/watch?v=YnFwyHfS67I

[2] GeeksforGeeks (2024). Uninformed Search Algorithms in AI. Retrieved from https://www.geeksforgeeks.org/artificial-intelligence/uniformed-search-algorithms-in-ai/

[3] Huddar, M. (2022). Breadth-First Search Algorithm Solved Example & Depth First Search Algorithm. Retrieved from https://www.youtube.com/watch?v=6B0mSyyFE2M and https://www.youtube.com/watch?v=cq0cBynFFXI

[4] GeeksforGeeks (2024). Heuristic Search Techniques in AI. Retrieved from https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/

[5] GeeksforGeeks (2024). A* algorithm and its Heuristic Search Strategy. Retrieved from https://www.geeksforgeeks.org/artificial-intelligence/a-algorithm-and-its-heuristic-search-strategy-in-artificial-intelligence/

[6] Huddar, M. (2024). AO* algorithm in AI - AND OR Graph. Retrieved from https://www.youtube.com/watch?v=9bjsOhSlwVM

[7] GeeksforGeeks (2024). Mini-Max Algorithm in Artificial Intelligence. Retrieved from https://www.geeksforgeeks.org/artificial-intelligence/mini-max-algorithm-in-artificial-intelligence/

[8] GeeksforGeeks (2024). Alpha-Beta pruning in Adversarial Search Algorithms. Retrieved from https://www.geeksforgeeks.org/artificial-intelligence/alpha-beta-pruning-in-adversarial-search-algorithms/

[9] Deeba Kannan. Artificial Intelligence - Min Max Algorithm. Retrieved from https://www.youtube.com/watch?v=J0Km3aOrDVE