

Optimal Path Finding

Unit No 4

Dr. Praveen Barapatre

Admissible A*

- In the realm of pathfinding algorithms, A* stands out as a remarkably efficient and versatile tool.
- Its effectiveness hinges on the quality of the heuristic function it employs.
- An **admissible heuristic** is a crucial component that ensures A* finds the optimal solution.

What is an Admissible Heuristic?

- An admissible heuristic, denoted as $h(n)$, is a function that estimates the cost of reaching the goal node from a given node n .
- It's considered admissible if it never overestimates the actual cost.
- In other words, $h(n)$ must always be less than or equal to the true cost $h^*(n)$.

Why Admissibility Matters

- An admissible heuristic guarantees that A^* will explore the most promising nodes first.
- By prioritizing nodes that are closer to the goal and have lower estimated costs, A^* avoids unnecessary exploration of less promising paths.
- This focused approach significantly reduces the search space and speeds up the algorithm.

Ensuring Admissibility

To ensure admissibility, it's essential to design heuristic functions carefully. Some common techniques include:

- 1.Relaxation:** This involves simplifying the problem by removing constraints or obstacles. The resulting relaxed problem can be solved optimally, and its solution cost can be used as an admissible heuristic.
- 2.Pattern Databases:** These precomputed tables store optimal solution costs for specific subproblems. By combining multiple pattern databases, more accurate and admissible heuristics can be derived.
- 3.Domain-Specific Knowledge:** Leveraging domain-specific knowledge can lead to highly informative and admissible heuristics. For example, in a grid-based map, the Euclidean distance or Manhattan distance between two nodes can serve as admissible heuristics.

Iterative Deepening A*

- Iterative Deepening A* (IDA*) is a graph traversal and pathfinding algorithm that combines the best aspects of iterative deepening depth-first search (IDDFS) and the A* search algorithm.
- It's particularly useful for problems where memory is a constraint, as it requires significantly less memory than A* while still guaranteeing optimal solutions.

How IDA Works:*

1. Initial Threshold:

1. Start with an initial threshold, typically the estimated cost to the goal node from the starting node, calculated using a heuristic function.

2. Depth-Limited Search:

1. Perform a depth-limited search, exploring nodes up to the current threshold.
2. At each node, calculate the f-value ($f(n) = g(n) + h(n)$), where $g(n)$ is the cost to reach the node from the start, and $h(n)$ is the estimated cost to the goal).
3. If a node's f-value exceeds the threshold, prune the branch.

How IDA Works:*

3. Iterative Deepening:

1. If no solution is found, increase the threshold to the minimum f-value of nodes that exceeded the previous threshold.
2. Repeat the depth-limited search with the new threshold.

4. Termination:

1. The algorithm terminates when a solution is found or when the threshold exceeds a predefined maximum value.

Key Advantages of IDA*

- Memory Efficiency: IDA* uses significantly less memory than A* as it only needs to store the current path from the root to the current node.
- Completeness: It guarantees to find an optimal solution if one exists, similar to A*.
- Simplicity: The algorithm is relatively simple to implement and understand.

Key Disadvantage

- Repetitive Node Expansions: IDA* may re-explore nodes multiple times, which can be inefficient, especially in large search spaces.

When to Use IDA*

- When memory is a significant constraint and optimal solutions are required.
- In problems with large search spaces where A* might run out of memory.
- When the heuristic function is accurate enough to prune many branches.

Recursive Best First Search

- RBFS is a search algorithm that combines the space-efficiency of depth-first search with the optimality of best-first search. It's particularly useful for problems where the search space is large and memory constraints are a concern.

- 1.F-limit:** This is the estimated cost of the best alternative path available from any ancestor of the current node. It acts as a threshold for exploration.
- 2.Backtracking:** If a node's F-value exceeds the F-limit, the algorithm backtracks to the parent node and updates the F-limit to the best alternative path.
- 3.Recursive Exploration:** The algorithm recursively explores the most promising child node, updating the F-limit as needed.

```
RBFS(node, F_limit):
    if node is the goal node:
        return node.path_cost # Solution found
    for each child of node:
        child.f = max(child.g + child.h, node.f) # Calculate child's F-value
        # Sort children by increasing F-value
        children.sort(key=lambda child: child.f)
    for child in children:
        if child.f > F_limit:
            return F_limit # No better path beyond this point
        # Recursively explore the best child
        alternative = RBFS(child, min(F_limit, child.f))
        if alternative < F_limit:
            return alternative # Found a better path
    return children[0].f # No better path found
```

How it Works

- 1.Initialization:** The algorithm starts with the initial node and an initial F-limit (usually infinity).
- 2.Node Expansion:** The algorithm expands the current node, calculating the F-value for each child node.
- 3.F-limit Check:** The algorithm compares the F-value of each child node with the current F-limit.
- 4.Recursive Exploration:** If a child node's F-value is less than or equal to the F-limit, the algorithm recursively explores that child node.
- 5.Backtracking:** If all child nodes have F-values greater than the F-limit, the algorithm backtracks to the parent node and updates the F-limit to the best alternative path.

Pruning the CLOSED List

In search algorithms like A* and RBFS, the CLOSED list is a data structure that stores nodes that have already been explored. This list is crucial for preventing cycles and ensuring that the algorithm doesn't revisit the same node multiple times.

While the CLOSED list is essential for traditional search algorithms, RBFS's recursive nature and F-limit mechanism can make it less necessary.

- F-Limit Pruning: The F-limit acts as a dynamic threshold. If a node's F-value exceeds the current F-limit, it's effectively pruned, regardless of whether it's in the CLOSED list or not.
- Recursive Structure: RBFS's recursive exploration ensures that only the most promising paths are explored. This inherent pruning mechanism reduces the need for a traditional CLOSED list.

Pruning the OPEN List

- While RBFS doesn't explicitly maintain an OPEN list in the traditional sense, it implicitly prunes the search space using the F-limit. This F-limit acts as a dynamic threshold, ensuring that only nodes with promising F-values are explored.

How RBFS implicitly prunes the OPEN list

1.F-Limit as a Pruning Tool:

- 1.When a node is expanded, its children are evaluated based on their F-values.
- 2.If a child's F-value exceeds the current F-limit, it's immediately discarded, effectively pruning it from the search space.

2.Recursive Exploration and Backtracking:

- 1.RBFS recursively explores the most promising child node, determined by its F-value.
- 2.If a dead-end is reached or a better path is found, the algorithm backtracks, pruning the current branch.

Conquer Beam Stack Search

Conquer Beam Stack Search (CBSS) is a technique that combines the efficient exploration of Beam Stack Search (BSS) with the divide-and-conquer paradigm. This combination aims to further improve the search efficiency, especially in large-scale problems.

How CBSS Works

1.Problem Decomposition:

- 1.The original problem is divided into smaller, more manageable subproblems.
- 2.This decomposition can be based on various criteria, such as spatial or temporal constraints.

2.Parallel Search:

- 1.Each subproblem is solved independently using BSS.
- 2.This parallel exploration can significantly speed up the search process.

3.Subproblem Integration:

- 1.Once solutions for the subproblems are found, they are combined to form a solution for the original problem.
- 2.This integration may involve conflict resolution and optimization techniques.

Key Advantages of CBSS

- **Scalability:** By breaking down large problems into smaller ones, CBSS can handle complex scenarios that might be intractable for traditional search algorithms.
- **Parallelism:** The parallel nature of the algorithm allows for efficient utilization of computational resources.
- **Flexibility:** The decomposition strategy can be tailored to the specific problem domain, leading to optimal performance.