AI Algorithms: Search Techniques and Game Implementation

Documentation

April 12, 2025

Contents

1 Mount Olimp - Path Finding with A* and IDA*

1.1 Problem Description

The Mount Olimp problem consists of finding optimal paths through a network of 24 nodes distributed around Mount Olimp. Each node has specific coordinates, and the edges between nodes have associated costs. The goal is to find optimal paths from a start node to one or more goal nodes using informed search algorithms.

We implement and analyze two primary algorithms:

- A* (A-Star) algorithm
- IDA* (Iterative Deepening A*) algorithm

The problem is defined with:

- A graph with 24 nodes representing locations
- Edges between nodes with associated costs (distances)
- A start node and one or more goal nodes
- A maximum number of steps to perform (n)

1.2 Heuristic Function Design and Analysis

1.2.1 Heuristic Function Definition

The heuristic function estimates the cost from a current node to the closest goal node. For this problem, we use the Manhattan distance as our heuristic:

```
1 def heuristic(self, node, goals):
2     return min(abs(node.x - goal.x) + abs(node.y - goal.y) for goal
       in goals)
```

Listing 1: Manhattan Distance Heuristic

This function calculates the Manhattan distance (sum of absolute differences in x and y coordinates) from the current node to each goal node, then returns the minimum value. When there are multiple goal nodes, the heuristic represents the estimated cost to reach the closest goal.

1.2.2 Proof of Admissibility

A heuristic is admissible if it never overestimates the actual cost to reach the goal. We need to prove that our Manhattan distance heuristic is admissible in the context of the Mount Olimp problem.

Theorem: The Manhattan distance heuristic is admissible for the Mount Olimp problem.

Proof:

1. In our graph, movement is constrained to paths along edges with specific costs.

2. The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is defined as $|x_1 - x_2| + |y_1 - y_2|$.

3. This represents the shortest possible path distance if we could move only horizontally and vertically at unit cost.

4. In our problem:

- Edges have costs that are proportional to or greater than the direct grid distance
- The actual path between nodes must follow existing edges, which can never be shorter than the direct Manhattan distance

5. Since our problem uses a graph where:

- Movement is restricted to existing edges
- Edge costs are always greater than or equal to the corresponding grid distance

6. Therefore, the Manhattan distance can never overestimate the actual cost along existing edges to reach the goal.

7. Furthermore, when there are multiple goals, by taking the minimum Manhattan distance to any goal, we ensure the estimate remains admissible since the actual cost to the closest goal cannot be less than this minimum estimate.

Therefore, our heuristic is admissible for the Mount Olimp problem, guaranteeing that A* and IDA* will find optimal solutions when using this heuristic.

1.2.3 Implications of Admissibility

Because our heuristic is admissible:

- A* is guaranteed to find the optimal (shortest) path to the goal
- The search may expand more nodes than with a more informative (but possibly inadmissible) heuristic
- The admissibility property is crucial for ensuring solution optimality

1.2.4 Consistency Analysis

A stronger property than admissibility is consistency (or monotonicity). A heuristic h is consistent if for every node n and successor n' with step cost $c(n, n')$:

$$h(n) \leq c(n, n') + h(n')$$

Our Manhattan distance heuristic is also consistent because:

1. The triangle inequality property of Manhattan distance ensures that the estimated distance from node n to the goal cannot be greater than the cost of moving from n to any successor n' plus the estimated distance from n' to the goal.

2. Since consistency implies admissibility, this further confirms that our heuristic is admissible.

Consistency provides additional benefits:

- A* will never need to reopen closed nodes
- The first path found to any node is guaranteed to be optimal
- Search efficiency is improved

1.3 Implementation Details

1.3.1 A* Algorithm Implementation

The A* algorithm is implemented as follows:

```
def a_star(self, start_index, scopeNodes, n):
    open_set = []  # Priority queue
    step_count = 0

    # Add start node to priority queue with f-value of 0
    heapq.heappush(open_set, (0, self.graph.nodes[start_index]))

    # Initialize path costs (g) and total estimated costs (f)
    g = {node: float('inf') for node in self.graph.nodes.values()}
    g[self.graph.nodes[start_index]] = 0

    f = {node: float('inf') for node in self.graph.nodes.values()}
    f[self.graph.nodes[start_index]] = self.heuristic(
        self.graph.nodes[start_index],
        [self.graph.nodes[goal] for goal in scopeNodes]
    )

    while open_set:
        _, current = heapq.heappop(open_set)  # Get node with
    lowest f-value

        if current.index in scopeNodes:
            # Goal node found
            pass

        # Explore neighbors
        for neighbor, cost in self.graph.get_neighbors(current):
            score = g[current] + cost  # New g-value through
    current node

            if score < g[neighbor]:  # If we found a better path
                g[neighbor] = score  # Update g-value
                f[neighbor] = score + self.heuristic(
                    neighbor,
```

```
                    [self.graph.nodes[goal] for goal in scopeNodes]
                )  # Update f-value

                # Add to open set if not already there
                if neighbor not in [item[1] for item in open_set]:
                    heapq.heappush(open_set, (f[neighbor], neighbor
    ))

        step_count += 1
        if step_count == n:  # Stop after n steps
            return open_set  # Return nodes in open set

    return None  # No path found
```

Listing 2: A* Implementation

1.3.2 IDA* Algorithm Implementation

The IDA* algorithm combines iterative deepening with A*'s evaluation function:

```
def IDA_star(self, start_index, scopeNodes, n):
    start_node = self.graph.nodes[start_index]

    # Initial threshold is the heuristic value of start node
    threshold = self.heuristic(
        start_node,
        [self.graph.nodes[goal] for goal in scopeNodes]
    )

    while True:
        # Perform depth-first search with current threshold
        result, next_threshold = self._dfs(
            start_node, scopeNodes, 0, threshold, n, []
        )
        if result is not None:
            return result  # Path found
        if next_threshold == float('inf'):
            return None  # No path possible
        threshold = next_threshold  # Increase threshold for next
    iteration
```

Listing 3: IDA* Implementation

The depth-first search helper function:

```
def _dfs(self, current, scopeNodes, g, threshold, n, path):
    path.append(current.index)
    f = g + self.heuristic(
        current,
        [self.graph.nodes[goal] for goal in scopeNodes]
    )

    if f > threshold:  # If f exceeds threshold, backtrack
        path.pop()
        return None, f  # Return f as next potential threshold

    if current.index in scopeNodes:  # Goal found
```

```
        return path, None

    min_threshold = float('inf')
    for neighbor, cost in self.graph.get_neighbors(current):
        if neighbor.index not in path:  # Avoid cycles
            result, next_threshold = self._dfs(
                neighbor, scopeNodes, g + cost, threshold, n, path
            )
            if result is not None:
                return result, None  # Path found
            min_threshold = min(min_threshold, next_threshold)  # Track minimum

    path.pop()  # Backtrack
    return None, min_threshold
```

Listing 4: DFS Helper Function for IDA*

1.4 Performance Analysis

1.4.1 Algorithm Comparison

Algorithm	Memory	Time	Complete	Optimal
A*	$O(b^d)$	$O(b^d)$	Yes	Yes*
IDA*	$O(d)$	$O(b^d)$	Yes	Yes*

Table 1: Algorithm Comparison (*With admissible heuristic)

where:

- b is the branching factor (average number of successors)
- d is the depth of the solution

The key difference is that IDA* trades time efficiency for significantly reduced memory usage, making it suitable for memory-constrained environments.

2 Nine Men's Morris Game Implementation

2.1 Introduction

Nine Men's Morris is a traditional board game. This document focuses on the implementation of an artificial intelligence system for playing this game.

2.2 Game Rules Overview

Nine Men's Morris is played on a board with 24 positions arranged in three concentric squares connected by lines. The game consists of three phases:

1. **Placement Phase**: Players alternate placing their 9 pieces on empty intersections.
2. **Movement Phase**: After all pieces are placed, players alternate moving pieces along lines to adjacent empty positions.
3. **Flying Phase**: When a player is reduced to only 3 pieces, they may "fly" pieces to any empty position.

When a player forms a "mill" (three pieces in a straight line), they may remove one of the opponent's pieces that is not part of a mill (unless all opponent pieces are in mills).

2.3 Heuristic Evaluation Function

2.3.1 Phase-Dependent Evaluation

A critical insight in our implementation is that different game phases require different evaluation criteria. The algorithm uses two distinct evaluation functions:

- **Placement Phase Evaluation**: Used when either player still has pieces to place
- **Movement Phase Evaluation**: Used when all pieces have been placed

2.3.2 Placement Phase Heuristic

During the placement phase, the evaluation function is:

$$\begin{aligned} Score_{placement} = & 4 \times (Pieces_X - Pieces_O) + \\ & 6 \times (Mills_X - Mills_O) + \\ & 3 \times (PotentialMills_X - PotentialMills_O) + \\ & (Mobility_X - Mobility_O) \end{aligned} \tag{1}$$

2.3.3 Movement Phase Heuristic

During the movement phase, the evaluation function is:

$$\begin{aligned} Score_{movement} = & 4 \times (Pieces_X - Pieces_O) + \\ & 6 \times (Mills_X - Mills_O) + \\ & (Mobility_X - Mobility_O) \end{aligned} \tag{2}$$

2.3.4 Heuristic Components

Each component of the heuristic captures a critical aspect of the game state:

1. **Piece Difference ($Pieces_X - Pieces_O$)**:
   - Represents the material advantage
   - More pieces provide more opportunities to form mills
   - More pieces increase mobility and board control
   - Losing pieces accelerates further loss as removal opportunities increase
2. **Mills ($Mills_X - Mills_O$)**:
   - Completed mills are the primary mechanism for removing opponent pieces
   - Mills represent strategic control of key board areas
   - Mills can be reused by moving a piece out and back in
   - Multiple mills create compounding advantages
3. **Potential Mills ($PotentialMills_X - PotentialMills_O$)**:
   - Situations where a player is one move away from forming a mill
   - Create forcing moves that restrict opponent options
   - Can lead to tactical advantages
   - Especially valuable in the placement phase when positions are being established
4. **Mobility ($Mobility_X - Mobility_O$)**:
   - Represents strategic freedom and tactical opportunities
   - Limited mobility makes a player vulnerable to forced moves
   - In the movement phase, restricted mobility can lead to a loss
   - Mobility becomes increasingly important as the game progresses

2.3.5 Weight Justification and Analysis

The weights in the heuristic function (4, 6, 3, and 1) were carefully chosen based on strategic principles and empirical testing:

- **Mills (weight 6)**: Receive the highest weight because they directly lead to removing opponent pieces, creating a compounding advantage. Each mill formed potentially reduces the opponent's material by one piece, which further reduces their ability to form mills.
- **Piece Difference (weight 4)**: Having more pieces than the opponent is significantly advantageous but less immediate than having mills. Piece advantage becomes decisive in the movement phase as it directly correlates with mobility and control.
- **Potential Mills (weight 3)**: Valued highly during the placement phase as they represent imminent threats. This encourages the AI to set up strategic positions that will lead to mills in future moves.
- **Mobility (weight 1)**: Given the lowest weight because, while important, it's generally a secondary consideration to material and mills. However, it becomes decisive in the endgame when pieces are few and mobility constraints can lead to forced losses.

2.3.6 Implementation of Potential Mills Calculation

A key feature of our heuristic is the efficient calculation of potential mills:

```
def _count_potential_mills(self, player):
    potential = 0

    for row in self._winning_rows:
        player_pieces = sum(1 for pos in row if self.board[pos] ==
    player)
        empty_spots = sum(1 for pos in row if self.board[pos] == '.
    ')

        if player_pieces == 2 and empty_spots == 1:
            potential += 1

    return potential
```

Listing 5: Potential Mills Calculation

This function:

1. Iterates through all possible mill lines
2. Counts player pieces and empty positions in each line
3. Identifies situations where a player needs just one more move to complete a mill

2.3.7 Terminal State Evaluation

The evaluation function also handles terminal game states with extreme values:

- If player X has fewer than 3 pieces: $Score = -1000$
- If player O has fewer than 3 pieces: $Score = 1000$
- If player X has no legal moves: $Score = -1000$
- If player O has no legal moves: $Score = 1000$

These values ensure that winning and losing positions are clearly differentiated from non-terminal positions, guiding the search toward favorable game outcomes.

2.4 Search Algorithms

2.4.1 Minimax with Alpha-Beta Pruning

The evaluation function guides the decision-making process through a minimax search with alpha-beta pruning:

2.5 Performance Analysis

2.5.1 Heuristic Effectiveness

The effectiveness of our heuristic can be measured by:

- **Evaluation accuracy**: How well the heuristic values correlate with actual win probabilities
- **Search efficiency**: How effectively the heuristic guides the search toward promising positions
- **Strategic alignment**: How well the heuristic captures the key strategic principles of Nine Men's Morris

Empirical testing shows that our weighted combination of material, mills, potential mills, and mobility effectively guides the AI toward strong play across all game phases.

2.5.2 Search Efficiency

With alpha-beta pruning, the search efficiency is significantly improved:

- With perfect move ordering, alpha-beta pruning evaluates only $O(\sqrt{b^d})$ positions, compared to $O(b^d)$ for regular minimax
- The heuristic function helps improve move ordering by identifying promising moves early
- Typical branching factor in Nine Men's Morris ranges from 5-20 depending on the game phase

Algorithm 1 Alpha-Beta Search

```
1: procedure AlphaBeta(depth, α, β, maxPlayer)
2:     if depth = 0 then
3:         return (evaluatePosition(), this)
4:     end if
5:     currentPlayer ← 'x' if maxPlayer else 'o'
6:     nextStates ← generateSuccessorStates(currentPlayer)
7:     if nextStates is empty then
8:         return (−1000 if maxPlayer else 1000, this)
9:     end if
10:    bestState ← null
11:    if maxPlayer then
12:        value ← −∞
13:        for each state in nextStates do
14:            (childValue, _) ← state.AlphaBeta(depth − 1, α, β, False)
15:            if childValue > value then
16:                value ← childValue
17:                bestState ← state
18:            end if
19:            α ← max(α, value)
20:            if β ≤ α then
21:                break                                   ▷ Beta cutoff
22:            end if
23:        end for
24:    else
25:        value ← ∞
26:        for each state in nextStates do
27:            (childValue, _) ← state.AlphaBeta(depth − 1, α, β, True)
28:            if childValue < value then
29:                value ← childValue
30:                bestState ← state
31:            end if
32:            β ← min(β, value)
33:            if β ≤ α then
34:                break                                   ▷ Alpha cutoff
35:            end if
36:        end for
37:    end if
38:    return (value, bestState)
39: end procedure
```

3 Conclusion

Both the Mount Olimp pathfinding problem and the Nine Men's Morris game demonstrate the critical importance of well-designed heuristic functions in artificial intelligence applications:

- For the Mount Olimp problem, an admissible Manhattan distance heuristic ensures optimal pathfinding while guiding the search efficiently
- For Nine Men's Morris, a phase-specific weighted heuristic captures the strategic nuances of the game and guides the AI toward strong play

The heuristic functions in both cases illustrate key principles of heuristic design:

- **Domain-specific knowledge incorporation**: Both heuristics leverage understanding of their problem domains
- **Efficiency**: Both heuristics are computationally inexpensive to calculate
- **Effectiveness**: Both heuristics successfully guide the respective search algorithms toward optimal solutions

These implementations provide a solid foundation for further enhancements and optimizations in both pathfinding algorithms and game-playing AI systems.