COMP30024 Artificial Intelligence: Assignment 1 Report

Jamie A.C Marks (student no. 1382410) and Joshua Munday (student no. 1453141)

The University of Melbourne, Semester 1 2025

Search Strategy Overview

We decided to use the A* (A Star) algorithm discussed in lectures to solve this problem. We made this design choice because A* is optimal (given that the heuristic is admissible), which means our algorithm will always return the best possible solution. We chose an informed search strategy (A*), since the board is fully observable and contains important information about the future of the game; an uninformed strategy would forego using such information, rendering it less efficient. To efficiently implement A Star, we used a min-heap which has an access time complexity of $O(\log(V))$ where V is the number of elements in the heap. In addition, we used a move cache (in the form of a dictionary) to remember moves after they were calculated, to avoid repeated computation.

Time Complexity:

Noting that $n =$ the number of rows in the board **(not total nodes)**

A*:

In the worst case, we will have to explore all the nodes on the board, and we must add each node to the heap which has time complexity: $O(\log(V))$. So, the total worst-case complexity is $O(n^2 \log(n^2)) = O(n^2 \log(n))$. Since A* needs to store the whole board, the space complexity of this part is $O(n^2)$. Notably, the average case performance of A* is generally much better than this, since a strong heuristic means A* will only explore feasible search-spaces, rather than all nodes.

Move Generation:

Since a single move sequence may be comprised of many similarly calculated jumps, finding the possible move cases is well suited to a recursive solution. We used a recursive algorithm to discover all the possible moves from a certain point on the board (expanding a node). In the worst case a single node expansion could lead to the recursive algorithm exploring all the cells' neighbours recursively which has a time complexity of $O(n^2)$, since processing each node is done in constant time. However, after each cell in the board has been explored the move cache is used on each move generation step, since this cache is stored as a dictionary the lookup time is $O(1)$. So, the total time complexity for all move generation is $O(n^2)$.

Assuming the typical case of an exploration with some vertical movement, the recursive depth of the algorithm is $O(n)$. So now we have that the total worst-case number of moves for a single cell is $O(5^n)$. Since the move cache may end up storing all possible moves, the space complexity is also $O(n^2 \times 5^n)$.

Heuristic evaluation:

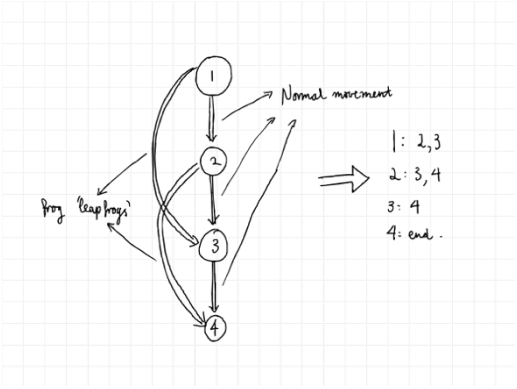
Our heuristic involves a modified Dijkstra search (a variant of best fist search) with time complexity$O(n^2 \log(n))$. However, this search only happens once over execution, so it does not dominate runtime (we discuss specifics later).

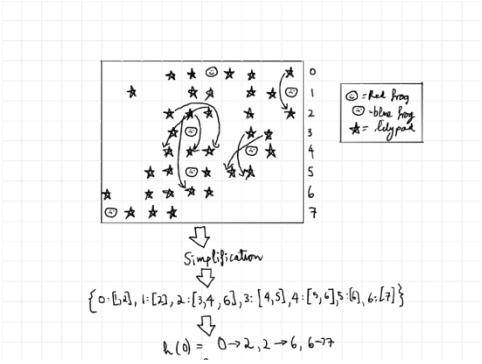
Total Complexity:

The A* search step dominates the time complexity. So, we have a total time complexity of $O(n^2 \times \log(n^2))$. The space complexity is dominated by the move cache and so the space complexity is $O(n^2 \times 5^n)$. However, we should note that the average case is closer to $O(n^2 \log(n))$, since this worst-case branching only occurs with many blue frogs.

2. Heuristic Choice and Demonstration

Our heuristic is based on a relaxed version of the game, where a blue-frog multi-jump can be accessed from anywhere on the row in which the multi-jump starts. We found all the possible multi-jumps in a precomputation step, by initiating moves from all orientations around each blue frog. Then, we retained only the information about which rows each move can move us between; we called these 'compressions' of a move.





This simplification allowed us to perform a graph-search on all 'compressions' to find the optimal sequence of multi-jumps and thereby the lower-bound on total required moves to finish – the condition of an admissible heuristic. Our graph-search was performed using Dijkstra's algorithm, where each node represents a row-height, and each edge represents a compression; we also added edges between each row and its direct successor (normal movement). This gives us an optimistic shortest path from the current row to the goal state. We performed the heuristic calculation as a preprocessing step, meaning we used Dijkstra's algorithm from each starting row, to avoid repeated computation.

Dijkstra has complexity O((V+E) log(V)) and has guaranteed optimality. If we assume both V and E are roughly comparable in magnitude to n (as is above), then each heuristic evaluation has a cost of $\approx n \log n$. Since we perform this search on each level, we get a total complexity of $n^2 \log n$. Combined with the finding compressions step (time complexity entirely dependent on arrangement of blue frogs) – we get an average precompute execution time of ≈0.01-0.1ms, with the upper bound coming from a board size of 25 x 25. Experimentally, we see a consistent 10-20% improvement in compute time, across most board sizes (between 8x8 up to 500x500), although the most consistent improvements come on large boards.

3. Extension to multiple red frogs

This would be a much harder problem to deterministically solve, since the branching factor becomes 6 times larger (since we can move any frog on a given turn). With that in mind, we might be better suited to a utility-based function, which estimates the optimal move, based on maximising forward movement, and minimising congestion (tightly clustered red frogs may interfere with one another, reducing optimality). Our solution would need a new heuristic (we would make it more complex, to account for the competing interests of different frogs), and a stricter end condition (since our terminal state is currently when a single frog reaches the end, whereas the new terminal state may require rearrangement of red frogs on the final row, and has multiple conditions that need to be met).