Single player Freckers discussion

Group 5x4MJ_E_0L4

Author: ZhenYiu Xu, YanRunZhe Lyu
Subject: COMP30024 Artificial Intelligence
Email: zhenyiux@student.unimelb.edu.au, yanrunzhel@student.unimelb.edu.au

Discussion:

As the first section of constructing an AI Frecker agent, this project required us to build an agent with a specific search algorithm to handle a simplified version of a Frecker game, moving a single red frog from the top row of the board to the bottom.

1.Among all the search strategies introduced, we precisely selected the A* search algorithm as the principal algorithm of the agent. The A* algorithm is optimal and complete in a finite scenario, which is suitable for finding the most efficient strategy to move the frog to the bottom on a finite 8x8 board. In the search.py file, we use the heapq data structure to store and pop explored positions with f and g values in A* search, since a heap can pop nodes with minimum priority efficiently in O(log n) time. A dictionary is used to store the minimum g value (cumulative distance from the start) of every explored position to expand nodes only with the lowest g value. To obtain the shortest path and movement directions, we use a list to store the parent node and the directions to the child node of each node in the path and retrieve the path at the end.

In order to handle the cases of successive jumps in a single move, we also introduced a depth-first search (DFS) algorithm to find a suitable position to stop. As each jump moves two units on the board and successive cross jumps can be made within one move, it is usually a favorable move to make. However, this does not imply that successive jumps are always beneficial for winning the game; hence, an additional search algorithm is required to find all possible cross-jump paths from the current position of the red frog (implemented in the get_cross_jumps function in search.py) and let A* decide the optimal path. In DFS, we backtrack after expanding nodes by using a list to mark visited nodes and a dictionary to store the directions to every landing point.

In terms of the search tree and state space, the theoretical time and space complexity of DFS would be O(b^x) and O(bx) respectively, where b is the branching factor of each node and x is the maximum depth of cross jumps. In every expansion process of A* search, it may invoke the DFS algorithm; hence, the overall time complexity of A* search is O(b^x + d), where d is the depth to the goal state, and the space complexity is O(b^d) as it keeps all nodes in memory. In this problem, the red frog can move in 5 directions, and the maximum depth is 8 due to the board size. Due to the actual board size and red frog movement, there can be a maximum of 6 successive cross jumps. Therefore, the time and space complexity of A* search can be O(5^14) and O(5^8). However, the space and time complexity are controlled due to the limited board size and legal movement in each step.

The successive cross jumps are likely to be dense in a singular path; hence, depth-first search is suitable as it has linear space complexity, and its exponential time complexity is restricted by the limited number of jumps it can make in practice. Theoretically, the space and time complexity would have a maximum proportional to the board size, but it is usually lower in practice.

2.In the A* search algorithm, we focused on using a heuristic based on the minimum Manhattan distance to the blocks with pads on the last row. It is calculated as min(|x - xi| + |y - yi|) for all i, where i are the blocks with pads on the last row. While acknowledging that movement in Frecker includes diagonal moves, the ultimate goal is to move to the other side, which makes the vertical distance essential for achieving the goal, and the horizontal distance determines whether a valid endpoint is on the path. Instead of considering the entire last row as endpoints, we only looked into the blocks with pads. This prevents potential heuristic errors that could mislead the agent towards non-valid ending positions. Setting up this heuristic eventually speeds up the search as it indicates the expected minimum number of steps each position requires to reach a valid endpoint and guides the search to explore nodes closer to the goal state first based on the heuristic. Manhattan distance is admissible since it gives the minimum step count to the goal state, which is always less than or equal to the true cost. Since A* search expands nodes in order of increasing f value, A* search is optimal for this problem.

3.Imagining the task is modified to require us to move all 6 frogs to the other side, the nature of this search problem would be heavily altered. The state would need to represent all 6 frogs, and the state space would grow exponentially. We would need to consider which frog to move in each action, resulting in a change to the state space. Due to the property that each pad is removed after being used once and assuming that growth stages are still prohibited, this imposes a strong limit on path selection and increases the likelihood of the problem having no solution. The search algorithm would need to be significantly altered to handle this situation, considering the depletion from the first frog to the last and constructing a non-repeating path for the frogs. This means the search algorithm would need to be reconstructed, as the most efficient path is probably not the one with the least pad consumption. Thus, we may need to use a more complicated heuristic to handle the interactions between the frogs if using a heuristic-based algorithm. One possible implementation is to leverage the property that a frog can jump over another frog of the same color and that successive jumps do not remove midway pads. Hence, we could construct an algorithm that plans the frogs into one line of successive jumps and moves them to the end positions one by one.