COMP30024 Artificial Intelligence
Project part A – Report

The assignment is based on implementing single player Infexion, in which our agent acts as the red player and tries to win in as few moves as possible, where red can only play SPREAD actions (SPAWN actions are prohibited). This translates into finding the shortest sequence of actions in order for red to win, given different board states as input.

We were able to implement A* search in order to achieve this successfully and were able to pass all the test cases. A brief overview of our strategy is outlined below.

Data structures

We used a heap-based priority queue to store the red nodes, with the priority calculated by summing the cost of the path taken so far and the heuristic value of the current node. Lists and dictionaries were used in various parts of the program, and a set was used to store the explored nodes.

Strategy and Heuristic function

As mentioned above, we were able to implement the informed search algorithm A* search with the use of a heuristic function based on Euclidean distance. Heuristic functions are used to estimate the cost of reaching the goal state from the current state, which informs the algorithm which node to explore next. The algorithm is more likely to choose nodes/states that have closer estimated distances to the goal state, which means we avoid searching the entire search space. Our heuristic is admissible as it always underestimates the true cost to the goal state since Euclidean distance is the shortest distance between 2 points, which is always less than the true distance to the goal node.

In our implementation, the heuristic() function calls function euclideanDistance() to obtain a list of distances between the current node we are exploring and each of the goal(blue), nodes and returns the distance to the closest goal node from the list. euclideanDistance() operates by iterating through each of the goal states and calculating the distance from the current node using Pythagoras' theorem.

The function runs a while loop until all the goal nodes are found (when red takes control of all tokens of the board). In each iteration of the loop, the function removes the current node with the lowest priority from the frontier queue.
If the current node is a goal state, the function records the path taken to reach the goal state and removes the goal state from the list of remaining goal states.
If the current node is not a goal state, the function generates the successors of the current node using the generateSuccessors() function. If the successor node has not already been explored or if it is not already between two goal states, we calculate the successors heuristic and cost and add each successor node to the frontier queue (populated with red nodes). We exclude nodes in between goal states to optimize finding more goals in one direction.

If we have found more than 1 goal in a direction, we remove all the goals found from the total_goals_found list except the last one as we only need to record the path to the last goal node. We use a list named 'path' to record the parent of the last node and append the parents until the root node to the end of the list. Therefore, we use the .reverse() function on path to reverse the items on the list, as we need the path from the initial node to the goal node.

Time complexity

The time and space complexity of A* search depends on the heuristic.

However, we believe it is in $O(b^{dg})$ in the worst case, where b is the maximum branching factor (here, b=6 as we expand nodes in the 6 hex directions), d is the depth of the tree and g in the number of goal states.

Space complexity

The space complexity of the algorithm depends on the size of the input board as well. However, since we are using a priority queue, we know that the space complexity is $O(b^d)$ in the worst case, with b and d as described above.

If SPAWN actions were possible:

It will decrease the time and space complexity as it will be more efficient and generates less nodes as nodes will only be generated for the spread functions and not for spawn.

Our code will be adjusted to only spread red nodes if the next move encounters a blue node. If a spread move does not find any blue nodes, our search function will iterate over every red node to try and capture blue nodes with spread. If there are blue nodes left, the next move is to spawn a red node next to the remaining blue nodes, then move again towards the blue nodes to capture it, recording the spawn location and direction taken to capture the blue nodes.

Justification of A* search

We chose to use A* search to solve this problem as it is more efficient in comparison to uninformed search algorithms like breadth first search (BFS). A* search expands fewer nodes as it is guided by the heuristic and is therefore a better choice overall. However, A* search can sometimes occupy more memory as we need to store additional information such as the heuristic values and the cost so far for each node. However, we believe this is a fair trade-off as the extra memory is used to make informed decisions, and A* is more likely to reach the goal state before BFS. Furthermore, A* generates less nodes than BFS, which makes A* more efficient in time, and since we are not searching an overly complex search space, memory is therefore not a restrictive factor in this context.