Weng Jae Chin (1255326)
Xin Yu Chuah (1255380)

# Project Part A Report

## Search Algorithm

A* Search Algorithm (Figure 1a), an informed search algorithm, was used to search the game tree. It determines which nodes to expand using an evaluation function of $f(n) = g(n) + h(n)$, where $f(n)$ is the estimated cost of the cheapest solution through $n$, the current node, $g(n)$ is the cost of getting to $n$, and $h(n)$ is an admissible heuristic function described in Question 2's answer. We chose the A* algorithm because it is complete, optimal and relatively straightforward to implement.

In terms of Data Structures, we created a Node Class (Figure 1b) in Python to represent the nodes; this can be found in `\search\node.py`. We also used a minimum priority queue to represent our search tree and to determine the expansion order. This is implemented using `heapq` from Python's built-in libraries. The minimum priority queue allows us to select the minimum estimated cost nodes to expand in $O(log\ n)$ time and inserting nodes into the queue in $O(log\ n)$ time. Both the time and space complexity of our A* search algorithm is exponential, $O(b^d)$, where $b$ is the branching factor of the search tree and $d$ is the depth of the solution; the same as typical A* search algorithms. The branching factor of our search tree is $b = 6 \times \#\, red\ pieces$; this is because each red piece on the board has the possibility of making a SPREAD action in 6 directions (Figure 1c). Additionally, the depth of the solution for our game will be equivalent to the cost.

## Heuristic Function

Our heuristic was derived from a relaxed version of the search problem, which allows every red piece to SPREAD up to 6 moves, even if the red piece did not have a power of 6. In this version, pieces do not need to converge up to a certain power in order to minimise cost to perform a SPREAD to reach a blue piece, as pieces can reach the goal independently at their own independent minimal cost.

The heuristic that was used is the minimum number of SPREAD actions required for a red piece to get to a blue piece. This heuristic is calculated by getting the Manhattan distance between the two pieces, which is also the shortest path between the two pieces, and dividing it by the maximum power each piece. A piece can only SPREAD in a straight line, so we also consider the cases where a red piece needs to change its direction to get to a blue piece. In these cases, we add 1 to the number of SPREAD actions required as the shortest path to another piece will only ever require at most one change in direction. Since using this heuristic we would favour game states that show red pieces being in positions that allow them to capture blue pieces in fewer steps, this would significantly speed up our search as the goal of the search problem is to have red pieces capture blue pieces and having them in this favourable position makes the capture faster.

This heuristic ensures the optimality of the solution as each piece cannot move more than 6 steps in one SPREAD action. This means that our estimated cost will not overestimate the cost of a red piece getting to a blue piece. Our heuristic function can be summarised by the function $h(x) = d(x)$, where $d(x)$ represents the minimum number of SPREAD actions from a red piece to a blue piece.

## SPAWN

If SPAWN actions were allowed, the branching factor of our algorithm will increase to $b' = 6 \times \#\, red\ pieces + \#\, empty\ cells$, due to the addition of possible SPAWN action nodes for every empty cell on the board. With no amendments to the A* search algorithm, the time and space complexity of the algorithm, given the new branching factor would be $O(b'^d) = O((b + \#empty\ cells)^d)$. For example, given the board in Figure 3a:

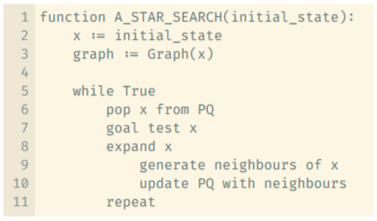
- only SPREAD: $O(b^d) = O((6 \times 1)^5) = 7,776$

- SPREAD and SPAWN: $O(b'^{d}) = O((6 \times 1 + 31)^{5}) = 69,343,957$

As such, even though $O(b'^{d})$ is also exponential, the addition of $\#empty\ cells$ can increase the time and space complexities of the algorithm significantly, especially if $d$ is large. However, it is also important to note that $d$ might decrease given the ability to SPAWN; for instance, given the board in Figure 3b:

- only SPREAD: $O(b^{d}) = O((6 \times 1)^{5}) = 7,776$
- SPREAD and SPAWN: $O(b'^{d}) = O((6 \times 1 + 34)^{2}) = 1,600$

Key modifications to our algorithm include: amending our board-state-updater (`update_board_states(state, action)`) to include the SPAWN action; generating SPAWN action nodes for each board state; adding a condition to decrease the number of SPAWN action nodes created (for example, favouring SPAWNs near the enemy); and possibly adding more features to the heuristic to differentiate between good and bad SPAWN actions.



```
function A_STAR_SEARCH(initial_state):
    x := initial_state
    graph := Graph(x)

    while True
        pop x from PQ
        goal test x
        expand x
            generate neighbours of x
            update PQ with neighbours
        repeat
```

Figure 1a: A* Search Pseudocode

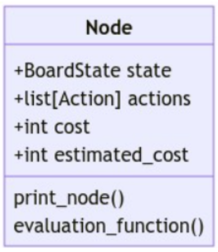


Figure 1b: Node Class

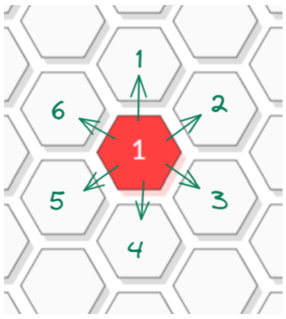


Figure 1c: SPREAD in 6 directions

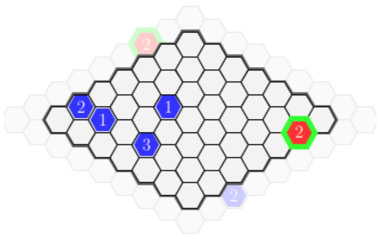


Figure 3a: Example board with 1 red piece, 4 blue pieces and 31 empty cells.

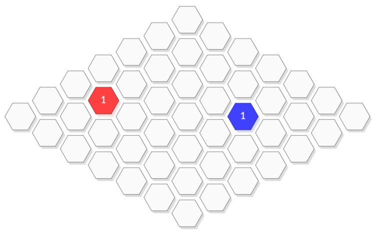


Figure 3b: Example board where having the ability to SPAWN decreases cost.