**1.** In this project, the A* search algorithm (as seen in the lectures) was used and implemented through the priority queue abstract data structure. Since pop and push operations are made frequently, a heap data structure was used, costing O(log n) for both operations.

The priority queue is initialised with the original board (with no spreads played initially). The priority for inserting a new board is the lowest estimated cost of the path from n to the goal: f(n) = g(n) + h(n). In case of a tie, we then prioritise based on the highest number of spreads played to get to the position. This is because if the total estimated cost f(n) is the same, but the number of spreads played is higher g(n), we are more likely to be closer to the solution if we explore that node ((h(n) is smaller). For each board popped from the priority queue, all boards that can be generated from spreading any red node in any of the six directions is considered. For each board inserted, the sequence of spreads played to get there from the initial position is kept track of.

The algorithm also keeps track of all unique board positions, so duplicate board positions aren't explored. This does not compromise optimality since the algorithm always prioritises boards in the queue with lower total estimated cost first and uses an admissible heuristic. If a board reaches one of the positions we have already visited, it must have either equal or higher total estimated cost f(n). Since the same board position always has the same heuristic value h(n) (same configuration of enemy nodes), it means that it took equal or higher number of moves g(n) to get to this same position, and hence it is not worth exploring.

The algorithm used ensures optimality of the search function. The first board found with no more blue nodes is guaranteed to be optimal. This is because the total estimated cost attached to that board f(n) (before popping from the queue and performing the spread) will be exactly equal to the actual cost it has taken to get to the goal after the spread. Since the A* algorithm always pops the lowest estimated total cost item from the queue, no other board will be more optimal than the solution found. In fact, f(n) will be exactly one greater than g(n) right before the winning spread since we make exactly one move at a time. (If h(n) = f(n) – g(n) is 2 or more, then we would need at least 2 moves to secure a victory, and if h(n) = f(n) – g(n) = 0, then we would have won already).

Time complexity: in the worst case, A* search can result in $O(b^d)$ time complexity, where b is the branching factor and d is the depth of the solution.

b, at each branching step, is approximately the number of red nodes * 6. However, unlike breadth first search where all branches and branches of branches etc are explored until the solution, using A* search algorithm means that not all branches added to the queue will be explored, so the number of branches not explored at each level grows exponentially as we explore down due to the recursive nature of the algorithm. This results in a significant reduction in the effective branching factor and improvement in the time complexity. The size of the improvement is correlated with how close the chosen heuristic is to the optimal heuristic for our A* algorithm.

b will on average be higher at each step if the initial board has a high number of red nodes and/or with a high power (k). Blue nodes with high power will also contribute to a higher power red node after being spread upon, as this can result in more red nodes as this higher power red node spreads (except a power 6 blue node, where the node is deleted after being spread upon). Blue nodes being further apart will also contribute to a higher average b value, since it will likely require red nodes to spread into more red nodes in order to reach it.

d, the number of moves it takes to find the solution, is dependent on the initial board state, and how far away blue nodes are from red nodes and/or from each other taking into account the power of the nodes.

Blue nodes with high power (excluding the max power 6) can become a high-power red node after a spread, which can be used to spread quickly to other blue nodes. Blue nodes being further away from each other may require a lot more moves to capture particularly in the case where a singular red node needs to capture them all.

Space complexity:
A* search keeps all nodes in memory, and in this case also keeps visited boards in memory to check for duplicates. This means that the space complexity will be $O(b^d)$

**2.** The heuristic h(n) used is based on the number of 'lines' in any of the 'r', 'q' or vertical directions it takes to clear all the enemy nodes. This will always be less than or equal to the actual number of spreads it takes to clear all the enemy nodes, since each spread can only clear a single line in one of the 'r', 'q' or vertical directions. Hence this is an admissible heuristic for the A* search algorithm. The cost so far g(n) taken to get to a position is simply the number of spreads taken to get there from the initial board.

The heuristic speeds up the search since these 'lines' give an indication of how many groups of blue nodes remain that can be spread upon quickly by red nodes.

Computing the heuristic of a board is done via a breadth first search algorithm. This is done by first initialising a queue data structure with the board, with an initial line count of 0. The costs for enqueue and dequeue are O(1). The algorithm selects an enemy node on the board and considers all three 'lines' in the 'r', 'q' and vertical directions, clears all the nodes in that direction, and adds the resulting board to the queue, incrementing the number of lines needed for this new board. Selecting a random enemy node does not impact optimality, since to clear that node, one of the three directional lines centred on that node will have to be part of the solution, since they are the only lines that can clear that particular enemy node.

Since breadth first search is used, the first board found with no more enemy nodes will correspond to the least number of lines needed to clear all of them from the original position, securing optimality.

For a given starting position, if no spawn actions are allowed, there are a finite number of configurations of enemy nodes that can be generated (from captures during spreads). The heuristic cost for a particular configuration of enemy nodes is constant. During the search process, since calculating a heuristic can be very costly if repeatedly called upon, the algorithm calculates and records each new configuration's heuristic cost in a dictionary. If the configuration has been seen before, then the algorithm can simply refer to the heuristic cost of that particular configuration without the need to re-calculate, significantly speeding up the search at the cost of some constant memory.

**3.** Spawns being allowed means that each step, the branching factor b will increase by the number of empty free nodes on the board, provided the total power on the board is within the allowed limit.

To accommodate this, all possible boards that can result from a spawn at each branching step need to be produced and added into the priority queue based on the associated priority of the board, keeping track of the sequence of spreads and spawns as well as total estimated costs for each position.