Q1

We used A* search algorithm with an admissible heuristic to guarantee optimality, which searches only to d, the depth at which an optimal solution can be found. There are 6 possible spread moves for each red cell in each state. Hence, the branching factor = 6r, where r = number of red cells on the board. In the worst case, when our admissible heuristic isn't efficient, the time complexity will be O((6r)^d) and space complexity will be O((6r)^d), since all nodes will be stored in memory up to depth d.

A* is implemented via a priority queue, here is the pseudo code:

- Given the root node, enqueue it to the priority queue.
- Continuously pop the top node from the priority queue and check if it's the goal. If it isn't, create and enqueue all its children. Each nodes' priority is calculated by the evaluation function.
- If the priority queue is empty and we still haven't found a goal state, then there isn't a solution.

Requirements for data structure: nodes are inserted & sorted one by one into the queue, hence we chose min heap for better time complexity in general.

- Min heap takes O(log n) time for insertion/deletion, which is faster than sorted array (O(n log n)) and sorted linked list (O(n)). All share O(n) space complexity.
- Access highest priority node in O(1) time

In each node, there is a pointer to the parent node and the move taken to go from the parent node to the current node. Hence, when we find a goal node, we can continuously trace back up the search tree to retrieve the moves taken to get to the goal node.

To check if a state is the goal, we check if the number of blue cells is 0.

Q2

Our heuristic is broken up into 2 parts, h(n) = h1(n)+h2(n).

h2(n) Calculation:

1. Take the highest powered red cell (e.g. power k) and spread it over the next k highest powered blue cells regardless of where they are on the board. (Which relaxes constraints)
2. Repeat step 1 until all blue cells are gone, let that be cost
    1. If no red, no blue cells left after calculation, then return h2(n)= 0
    2. Else if, some red red cells but no blue cells left, then return h2(n)=cost
    3. Else, when only blue cells are left, return h(2)=infinity

h1(n) calculation:

Define h1-type spread: the tokens in a token stack can be placed in any adjacent cell of the previously allocated cell. Which is a relaxation of the constraint of a linear spread.

1. If only blue cells are left, return +inf. If no blue cells are left, return 0.
2. For each red blue pair, find (closest distance - 1). Define this as P.
3. If red power > P, then a h1-type spread will kill off at least 1 blue cell. Return h1(n)=0.
4. If red power <= P, then P = 1 + (old P - red power), which is the cost of a spread plus the distance remaining to the adjacent cell of the blue cell
5. Find the pair that has the smallest P, and return h1(n)=P

h(n) Conclusion:

H1 and H2 calculate different parts of the process. Suppose we are given Fig1.

- h1(n) = minimum cost to rearrange the board (from Fig1 to Fig2) for h2(n)
- h2(n) assumes a new board (Fig2) such that it will require the least number of steps to reach the goal, compared to any other board configuration with the same tokens

Hence, their sum h(n) won't overestimate the true cost to the optimal solution.

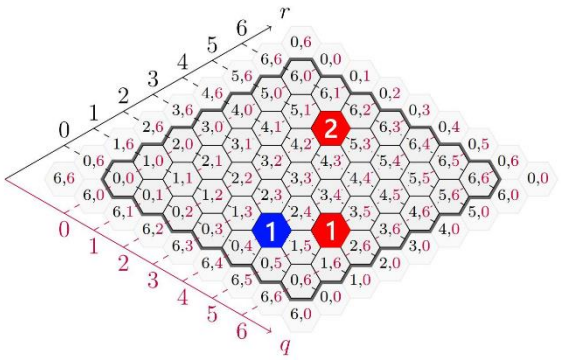

Figure 1: Current board

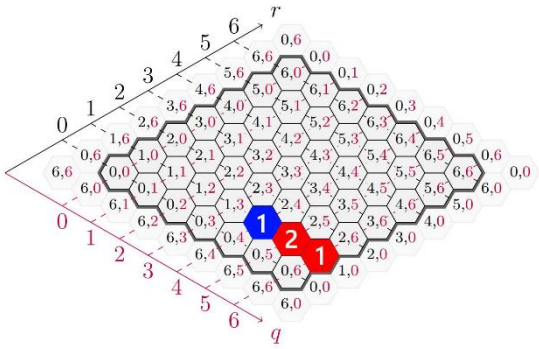

Figure 2: Rearranged board: least cost to goal state

In Figure 1, h(n) = h1(n) + h2(n) = min(2, 1) + 1 = 2.

This heuristic speeds up the search by giving higher priority (lower cost) to spread moves which move red cells closer to blue cells or moves which start eating blue cells. In comparison, brute force IDS method treats spread moves in all directions as the same priority, even if it moves red cells further away from blue cells.

Q3

If spawn actions were allowed in single player Infexion, this would increase the complexity of the search problem by increasing the number of possible moves from any given state. The new possible moves include spawning a new red cell in any empty cell on the board, given we don't go over 49 total power.

This translates to increasing the branching factor.

- Old branching factor = 6r, where r = number of red cells on the board
- New branching factor = 6r + e, where e = number of empty cells on the board

Our solution would need to modify the following:

- Create additional children nodes per node, to take into account spawn actions.
- A board with no red cells but at least one blue cell is no longer a dead end state
    - h1(n) for this state returns 1 (cost of spawn) instead of infinity
    - h2(n) for this state still returns infinity
    - Overall the h(n) evaluated for this state is still infinity, but children can now be generated from these states

Strategy modifications:

- To compensate for the increase in branching factor contributed by the number of empty cells the spawn action considers, we restrict the spawn locations considered only to the empty cells adjacent to blue cells.

Complexity:

- Time complexity will be O((6r+e)^d)
- Space complexity will be O((6r+e)^d)