Project A
Introduction to AI

Abris Gilvesy, Student ID: 1435563
Yoav Javits, Student ID: 1432134

Question 1: Strategy

First, we implemented a standard BFS algorithm to serve as a benchmark for optimality and speed. The BFS is implemented using the NodeBFS class. An instance of the class contains information about a given tile's position, its parent (i.e. which NodeBFS instance has it spread from), the state of the board, the power of the tile, the offset of the node relative to its parent, and a list of visited nodes.

The concrete implementation is as follows. We initialize a queue with all the red nodes. We enter a loop that runs until the queue is empty. We deque the first node on the list and check if it is the goal state. If it is, we return the optimal sequence of actions using the parent node saved in each NodeBFS instance. Otherwise, we calculate all the neighbors of the node, keeping in mind that the power of a node influences the number of neighbors. We iterate through all the neighbors and check if the power of the potential new node exceeds 6 or already has been visited. If yes, that node is disregarded. Otherwise, the state is updated according to the rules of the game, and saved in a new node. The new node is added to the queue and the visited list (of the new node).

The time complexity of the BFS algorithm is O(b^d) where b is the branching factor and d is the depth of the optimal solution. The upper bound on the branching factor is 288 because there can be 48 different red cells and each red cell can be expanded in 6 directions at each iteration. At each iteration, we store a constant quantity of data, and therefore our space complexity is likewise O(b^d).

Next, we implemented A* search, adapted to handle multiple sources and multiple targets. Our A* algorithm is implemented using the Node class. It has all attributes of NodeBFS along with three additional fields: g, the number of steps to reach the current node, h, the heuristic estimation of the number of steps from the node to the goal, and f, the summation of g and f. The BFS and A* algorithm implementations are identical save for three critical differences: A* expands the node with the lowest f attribute at each iteration, a heuristic value is calculated and saved for each new child, and goal checking is performed at child generation, not visitation.

Question 2: Heuristics

We attempted to use the following heuristic: h = max(axial_distance(a, b) - k + 1, 1), where a is the coordinates of the node that the heuristic is being calculated for, b is the coordinate of a blue node, and k is the power of the current node. Importantly, this calculation was performed for all blue nodes and the minimum value was assigned as the h attribute of the current node. In plain English, the heuristic is an underestimate of the number of steps to reach the closest blue node from the current node. The axial_distance is the Manhattan distance generalized to a hex grid and subtracting k (and adding one) ensures it is at most equivalent to the number of steps to reach the closest blue node. Importantly, it is an underestimate in the event that a given blue node is within k tiles of the current node, but not in one of the 6 principal directions. Finally, the max() function prevents negative moves.

The heuristic is admissible because the number of steps to reach the closest blue node is necessarily less than or equal to the number of steps to reach all the blue nodes. Since the heuristic underestimates the steps to reach the closest blue node, it remains admissible.

At the time of submission, our A* algorithm implementation was quick, but unfortunately, not 100% optimal. Therefore our solution uses BFS.

Question 3: SPAWN Actions

The search space will increase as the number of possible moves from each state is increasing massively. A player may SPAWN a token in an empty cell, on the condition that the total POWER of cells on the board is less than 49. That generates a lot of new possibilities for expansion, which increases the branching factor, which of course increases the time and space complexity.

One way to modify our solution to a situation when SPAWN actions are allowed is to spawn red cells in every empty cell and put them at the end of the queue (for BFS) or calculate the heuristic and put them in the priority queue (for A*). This is not the optimal solution, because it will increase the branching factor the most, and therefore will massively increase the time and space complexity.

Another way to modify our solution is according to the following observations - if there isn't any blue node that can be reached by any red node after a maximum of 2 iterations, we can SPAWN a red node next to one of the blue nodes that have some empty cell neighbor, and then SPREAD the red node to the blue one. This observation can change our heuristics and make them more dominant, therefore improving time and space complexity.