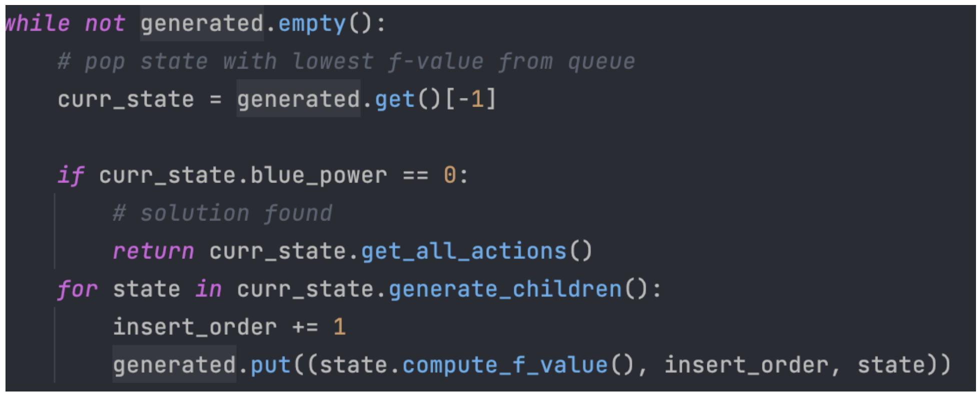
1. Our team implemented an A* search strategy. We used a priority queue as the data structure to enqueue and dequeue the leaf states. The “priority” used is the f-value of the state being enqueued, which is the estimated total cost of the path of the solution from the initial state, through the current state, to the goal state. Most of the complexity for this search comes from the computation of the f-values for the children of the current state that is popped from the priority queue, which we have to do when we generate the children of the current state.

   The calculation for the f-value of a given state involves the addition of the cost of the path of the solution from the initial state, which we called the g-value, with the cost from the current state to the goal state, the heuristic. We added the g-value as an attribute for our board_state class, so retrieving this is an O(1) operation. The calculation of the heuristic involves a nested for loop, in which the outer for loop iterates through the list of blue cells on the board, and the inner for loop iterates through all the items on the board. The time complexity of the inner for loop is O(n), where n is the number of items on the board. The time complexity of the outer loop is O(1), as the number of blue cells does not change. Hence the overall time complexity of the heuristic function equates to O(n), as we can ignore constants in big O notation.



```
while not generated.empty():
    # pop state with lowest f-value from queue
    curr_state = generated.get()[-1]

    if curr_state.blue_power == 0:
        # solution found
        return curr_state.get_all_actions()
    for state in curr_state.generate_children():
        insert_order += 1
        generated.put((state.compute_f_value(), insert_order, state))
```

   In the above for loop, insertion into a priority queue is O(log m), where m is the number of items in the priority queue, and since the queue is used to store the number of generated states we have, this becomes O(log b^d), where b is the branching factor and d is the depth of the least cost solution. In addition, we calculate the f-value for each child, an O(n) operation. The for loop will iterate over all children of a state, and the maximum number of children a state can have is equal to the branching factor b. Thus the total complexity of the for loop equates to O(b log m + bn).
   The while loop will iterate for a total of b^d times in the worst case. Summing all the complexities up, the worst case time complexity of the search function is O((b^d)*(b log b^d + bn)).
   The space complexity of the search function will depend on the priority queue, and the size of the board. The priority queue is used to store the generated children states, so the priority queue will depend on the branching factor b. We will have at maximum b^d states in the priority queue at any time, where d is the depth of the least cost solution. The space complexity of the board state will consist of the space taken up by the attributes. The most costly attribute is the board attribute in the board_state class, which takes up O(n) space, as it stores information about the n cells on the board. The remaining attributes take up a constant O(1) space, so the board_state class has a total space complexity of O(n). Thus the total space complexity of the priority queue equates to O(n*(b^d)). As no other functions take up space in search function, the total space complexity of the search is O(n*(b^d)).

2. The heuristic used involved keeping track of the coordinates of the blue cells using a list, and estimating the sum of the costs to take each blue cell. We believe that this heuristic speeds up the search by 'encouraging' red cells to spread towards blue cells, as it would reduce the estimated cost.
   For each blue cell, the program goes through the list of red cells on the board to find the red cell that requires the minimum amount of steps to get to that blue cell, and stores (coordinates of red cell, estimated cost for red cell to spread to the blue cell) as a key-value pair in a dictionary.
   The cost of spreading from a red cell to a blue cell is estimated using the formula:

   cost = max(0, min_distance - power) + 1

   i.e. the initial spread covers [power] number of cells, and each remaining cell requires 1 step to spread to

   Where

   power is the power of the red cell

   min_distance = min(x_diff, y_diff) + abs(x_diff - y_diff)

   i.e. the distance assuming the red cell can spread to any of the eight cells surrounding it

   Where

   x_diff = abs(blue_x - red_x)

   y_diff = abs(blue_y - red_y)

   In the end, the sum of the costs in the dictionary is returned as the heuristic.

   If two or more blue cells share the same red cell from which the least cost is achieved, the maximum least cost is kept in the dictionary. This guarantees that the estimated cost is always equal to or less than the actual cost, since the estimated cost is the best-case scenario, when all blue cells sharing the same least-cost red cell are eliminated in the process of spreading the red cell to the max least-cost blue cell. In other scenarios, the actual cost would be larger. In addition, the assumption that the red cell can spread in 8 directions when estimating the distance also makes the estimated distance lower or equal to the actual distance. Therefore, the heuristic is admissible, and hence the A* search is optimal.

3. If spawn actions are also allowed, it would increase the branching factor of the search tree, as the player now has more actions to choose from in each state of the game, thus increasing the complexity of the search. However, since the spawn action reduces the cost of getting close to the blue cells, we can take advantage of this and develop a strategy that finds a solution that takes fewer number of actions, which might also have a lower time complexity, since the search depth would be lower, depending on the initial board state.

   For example, when estimating the least cost to get to each blue cell, if the estimate is larger than 2, we can decide to just spawn a red cell next to that blue cell, which would require 2 actions in total to eliminate that blue cell. Implementing this would limit the depth of the least cost solution, d*, to 2 x number of blue cells, making the time complexity of the search function O(b*^d*), where b*=b + number of empty cells. Since b* >= b, d* <= d, the complexity of this search is lower than the original search in some cases.