Artificial Intelligence Report Part A

Question 1

Implementation:

- For each hexagon, the algorithm will record the cost required to reach it and its predicted heuristic value.
- Starting from the original hex, A* search will generate all possible neighbours and calculate their 'function' value, which is the **cost + heuristic** from the current hexagon. The parent hexagon of each neighbour will also be recorded.
    - Data will be stored in separate dictionaries, with the hexagon's coordinates as the keys, which map to their respective cost, function value, and parent hexagon. (Hence the 3 dictionaries).
    - The cost, function value and parent dictionaries will be reassigned **only if** the hexagon's current cost function is less than its previous cost.
    - Each hexagon will also be added to 'open', to keep track of which hexagons have **not** had their children explored fully. 'Open' is a set to ensure there are no double ups.
- Remove the hexagon with the **lowest** recorded function value from 'open', and repeat step 2 **with this hexagon** (instead of the starting hexagon).
    - **If the this hexagon is the goal hexagon**, then use the 'descended_from' dictionary to **return** the **path**, and the 'g_cost' dictionary to **return the cost** (optimal).
- If there are no more hexagons to remove in 'open', all nodes have been explored, so no solutions are possible: **return 0.**

Note: b = branching factor, d = depth of tree

The worst-case time complexity is O(b^d), where every neighbour has 6 other neighbours. However, given there are obstacles and edge hexagons, it is unlikely that every neighbour has 6 neighbours. The heuristic, with a reasonably tight lower bound, will significantly reduce the branching factor (as it eliminates neighbours which are predicted to be more costly). The space complexity will be impacted in a similar way, as each node that is traversed will be held in memory. This results in O(b^d) space complexity, with the potential for significant improvements with a heuristic with a tight lower bound.

Question 2

The heuristic function takes the board size, the current hexagon, and the goal hexagon as arguments. It ignores the existence of any hexagons which have been 'captured', and then calculates the minimum number of additional tiles required to get from the current hex to the goal hex.

The heuristic is calculated depending on two different scenarios.

1. If direction from current to goal is top-right, bottom-left or along the 'r' or 'q' axes, then **cost** = **sum(difference(current hex 'r', goal hex 'r'), difference(current hex 'q', goal hex 'q'))**.

2. Otherwise (if the direction is top left or bottom right), the cost is calculated by taking the maximum of the differences between the current and goal hex's 'r' and 'q' coordinates respectively.

This will return the minimum cost required to get from the current hex to the goal hex, given there are no obstacles. Hence, $h(n) \leq h^{*}(n)$, where h(n) is the estimated cost from hexagon 'n' to the goal, and h*(n) is the *true* cost. This is because the true cost can either be **equal to** the minimum cost (which is provided by the heuristic), or greater than the minimum cost if there are obstacles blocking the optimal path, forcing a longer route. Thus, it is admissible.

Because the heuristic consists only of if/else statements and **no** loops/recursion, the heuristic is in O(1) time. As the heuristic is a reasonably tight lower bound on the actual cost, this reduces the branching factor, hence, significantly improving the worst-case scenario time discussed in Q1. The closer the heuristic is to the actual cost, the more branches will be eliminated, and the time complexity will tend to a polynomial function.

Question 3

Heuristic will no longer be admissible. For example, if there are pre-existing red hexagons on the board which are included in the optimal path, the red player can utilise these pre-existing hexes without extra cost. Hence, using the original heuristic, the predicted cost can be greater than the actual cost in this scenario.

Instead, we should extend our previous heuristic to calculate the distance to the "occupied hexes" that can improve our path cost. The heuristic should predict the cost using the heuristic function derived in Q2, following these steps:

Note: A hexagon 'chain' is 1 or more usable hexagons that are connected already on the board.

1. Heuristic from current to goal (defined as H1)
2. Heuristic from current to an occupied hexagon chain of the same colour **plus** the heuristic from the closest hex (to the goal) in that chain, to another occupied hexagon chain or the goal. Then, continue adding more neighbours **until goal is reached or the running sum exceeds H1.)**
    - Repeat this process with all possible "occupied hexes" on the board (it will be a finite value), and not too lengthy because as soon as the running sum exceeds **H1**, it will terminate and go onto the next calculation.

After these calculations have taken place, the heuristic function will return the **minimum value calculated above**.

Using this heuristic on the same algorithm which ignores all obstacle hexagons ensures that the optimal solution will be found.