Algorithm

The algorithm implemented is the **A*** search algorithm.

A list of nodes is maintained, where each node stores the following information:

- **Position:** the current game state represented as a FEN string.
- **g-cost:** the actual cost from the start state to the current state.
- **h-cost:** the estimated (heuristic) cost from the current state to the goal state.
- **f-cost:** the total estimated cost `f = g + h`.
- **Parent index:** the index of the parent node in the list.
- **Move:** the move taken to reach this node from its parent.

A **priority queue** is used to select the node with the lowest `f-cost` (and lower `h-cost` as a tiebreaker).
Additionally, an **unordered map** is used to record whether a position has been visited and to store its corresponding node index. This prevents redundant state expansion and reduces computational overhead.

The algorithm proceeds as follows:

1. **Initialization:**
    - Start from the given position and push it into the priority queue with `g = 0` and the calculated `h`.
2. **Main Loop:**
   While the priority queue is not empty:
    - Check if the total runtime exceeds **10 seconds**. If so, terminate and output `-1`.
    - Pop the node with the smallest `f-cost`.
    - If this node's position represents a **winning state** (all opponent pieces captured), reconstruct the move sequence by tracing back through parent indices, then output the total cost and move list.
    - Otherwise, generate all valid moves from the current position.
3. **State Expansion:**
   For each valid move:
    - Apply the move to produce a new position.
    - Compute its `g` (current `g + 1`) and `h` (via the heuristic).
    - If this new position has not been visited, or if it can be reached with a lower cost, create a new node and push it into the priority queue.

If the queue is exhausted without finding a goal, output `-1` (though this case should not normally occur).

Heuristic Function

Design

The heuristic estimates the number of steps required for all **black pieces** to capture all **red pieces**.
For every red piece on the board:

1. Find the **nearest black piece** capable of capturing it.
2. Compute the minimum number of steps needed for that black piece to reach and capture the red piece:
    - For most pieces, this is the Manhattan distance between the two squares.
    - For **Chariots**, the cost is 1 if the target lies on the same rank or file (i.e., reachable directly), otherwise 2.
3. If multiple red pieces are competing for the same black piece, adjust the heuristic as follows:
    - If the same black piece has already been assigned to another red piece, combine their distances intelligently to reflect potential sequential captures.
4. If a red piece cannot be captured by any black piece (e.g., all black pieces are weaker), a constant large penalty (20) is added.

The sum of these minimal estimated capture costs over all red pieces forms the heuristic value.

Proof of Admissibility

The heuristic is **admissible** because it never overestimates the true cost to reach the goal.

- Each red piece's contribution to the heuristic is based on the **minimum possible distance** to a black piece capable of capturing it.
- When multiple reds share the same black piece:
    - If one red is **closer**, the heuristic assumes the black will capture it along the way to the farther red — thus **no extra cost** is added.
    - If both reds are **equidistant**, the heuristic assumes they lie on the same line (rank, file, or diagonal), allowing capture in one or two steps (as in the example test case 1-1).
    - If the current red is **farther**, the heuristic updates the maximum distance for that black piece, representing an optimistic (ideal) case where the nearer red will be captured en route.

Because all these assumptions represent the **best-case capture sequences**, the heuristic always provides a lower bound on the true cost. Hence, it is admissible.