Q1 How have you formulated the game as a search problem? (You could discuss how you view the problem in terms of states, actions, goal tests, and path costs, for example.)

Since the current stage of this game is single player only, the quickest way to successes would be moving all your chess pieces to its destination with minimum steps. Therefore, this stage can be interpreted as a problem of search for shortest path.

The hive-like chess board consists 37 hexagonal positions, we used the coordinate system provided in the Assignment spec.

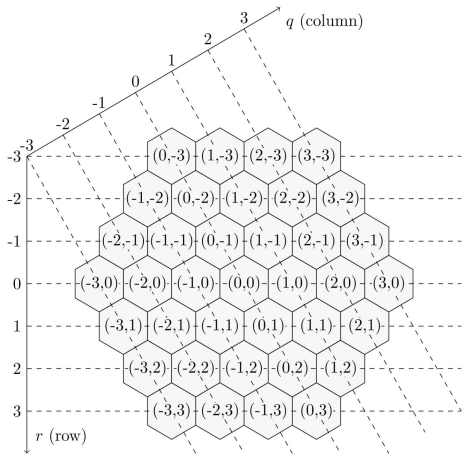

Figure 1: The q and r axes, with (q, r) indices for all hexes.

We consider the existence of pieces and blocks on the board to be a “state”. The state follows the same format as argument board_dict of function print_board in skeleton code, which is a dictionary with tuples(index) for keys and colour (or ‘block’) as values.

We discovered that the boundaries of this board can be expressed as functions, such as q = 3 (red destination), r = 3 (green destination) and q + r = -3 (blue destination).

We consider any action the target piece performs to be at cost 1, no matter it’s a move or a jump. Each piece has six directions to move, which can be expressed as: (q, r) + [(-1, 0), (0, -1), (1, -1), (1, 0), (0, 1), (-1, 1)]. Since there’s only one exit action for each piece, and only occurs at the end of its actions, we ignore it until the output stage.

Q2 What search algorithm does your program use to solve this problem, and why did you choose this algorithm? (You could comment on the algorithm’s efficiency, completeness, and optimality. You could explain any heuristics you may have developed to inform your search, including commenting on their admissibility.)

Compare to Dijkstra's Algorithm, BFS and DFS, A* is superior in this case, because it saves memory by not expanding all children nodes. Considering we have jump action in this game, a heuristic that only considers move actions could sometimes be greater than the real cost.

We created a variable called offset, it is used to make the heuristic closer to real heuristic. We add weight to offset if blocks on the path are clustered, minus weight if blocks are spread out, because a piece can jump more on a more spread out layout.

Q3 What features of the problem and your program's input impact your program's time and space requirements? (You might discuss the branching factor and depth of your search tree, and explain any other features of the input which affect the time and space complexity of your algorithm.)

Since our programme does not have a block wall detection design, it can't produce a reasonable heuristic under the circumstance shown below (the 30MOVE.json on LMS discussion board provided by **Akira Wang**). Our project show poor performance in terms of both space and time on this case, since our programme produces too many unnecessary children nodes.