Report

1. The game is a two-dimensional array of hexagon cells that agents with a characteristic of upper tokens can move from cell to cell under two methods. There are three kinds of objects which are upper tokens, lower tokens and blocks. Each state in this game shows which objects are in which cells.
   This game can be formulated as a search problem by upper tokens finding and taking the best next action among their next available actions, record their action sequence, update the state and keep checking if they reached the goal by doing goal tests. Ideally, path costs will also be considered to help find the optimum solution with the least sequence of actions needed.
   State: We used a dictionary called state to record the current state that contains the coordinates of all the objects on the board, it will renew after each upper token makes a movement decision.
   Actions: Each upper token can take either SWING, or SLIDE to move to a new cell. As the cells are hexagons, there are six directions on the board. Ignoring other conditions, under SLIDE method, each upper token can move to the six surrounding cells. If an upper token connects to another upper token, it can use SWING to move to either of the three corresponding opposite cells. In total, there are up to twelve possible actions an upper token can consider for each turn.
   A dictionary called open_close_dict in the format of {(upper token1): [[open list1], [close list1]], (upper token2): [[open list2], [close list2]], …..}is used for storing all possible next actions in open list and their sequence of actions in close list for all upper tokens.
   Goal tests: The game will stop if the last lower token is defeated by the upper token, thus the goal state should be no lower token on the board. In our code, we used a sorted_goal_dict to store the defeatable lower tokens for each upper token. When a lower token is defeated by an upper token, it will be deleted from the sorted_goal_dict. If all the defeatable lower tokens for an upper token were deleted, the upper token key will be deleted from sorted_goal_dict. Thus, if there is no lower token on the board, the sorted_goal_dict should be empty. Testing whether the sort_goal_dict is empty can be used to check if we are in a 'goal state'.
   Path costs: Each action costs 1 and the path costs should be the number of movements each upper token has taken by arriving at the goal state. (In our implementation, we used Greedy Search and haven't considered path costs.)

2. **What algorithm:** We used the Greedy Search algorithm to solve the problem.
   Why: The most important thing while implementing a search algorithm is the queueing function. As it defines the search strategy by picking the order of node expansion and the way of storing new leaf in the queue. Because we try to find the path with the least cost for each upper token, the queueing function for the searching algorithm should be a priority queue with a priority of finding the closest distance to the current target lower token among all available next moves.
   The greedy algorithm fits our method of solving this project. As a special case of Best First Search, it uses an evaluation function which is the heuristic to estimate its desirability--in our case, it is the distance between the next available move and the lower token they are going to defeat(goal). Our implementation ignores the details in between and assumes a straight line of distance which is also the case for Greedy Search. In addition, Greedy search expands the most desirable unexpanded node, which is similar to us moving to hex with the lowest cost(closest) to lower token(goal) and then exploring all possible next moves from that hex. And greedy search also supports us using a priority queue to find the shortest distance.

   Optimality: Greedy search is not optimum, it cannot find the optimum solution and only estimates the future cost which is not accurate enough. Also, it doesn't count the past cost. Hence it is not trying to find the shortest path.
   Completeness: In general, Greedy search is not complete, as it may be stuck in infinite loops. In our implementation, it is close to complete. We removed the next possible move if that upper token has already visited that hex before. Our game made it finite space with repeated state checking.

Efficiency: Greedy search is not efficient. It's better than an uninformed search as it looks at the future cost. However, the performance of Greedy Search depends on the heuristic it chooses. If the heuristic is lousy, it may need to look at all nodes. If the heuristic is good, then time could improve a lot.

- **Time complexity:** If we measure the branching factor (possible movements) as b, and the search tree depth (path length) as m, in the worst case the time complexity will be O(b^m).
- **Space complexity:** For greedy search, the worst case space complexity is also O(b^m). But a good heuristic could improve it. In our case, we measure the number of upper tokens as U, a number of lower tokens as L, and the number of total tokens as T. We used a state dictionary to store states with space complexity $O(T)$ and an open_close_dict dictionary to store possible movements and path history for all upper tokens with space $O((b + m)U)$, as they will renew after each movement, the total space complexity will be $O((b + m)U + T) \approx O(m)$which means it depends on the depth of the tree.

Heuristic: It estimates the future cost only. In our case, it is the Euclidean distance between the current token's possible next move and its goal.

Admissibility: As the heuristic is always <= true cost, it is admissible. It ignores all details in between and is not very accurate. However, it is a guarantee to be <=true cost as it is a straight line distance.

3. **Number of upper tokens > Number of lower tokens in the starting configuration:**

In our algorithm, we assign each upper token with a different lower token as a goal at first. If an upper token isn't assigned with any target, then it will move randomly under a random_move function.

- **Time complexity:** The time required to get to the goal state is still the time needed for the upper token to defeat the last lower token, the search tree depth does not change. Thus, in the worst case the time complexity is O(b^m).
- **Space Complexity:** For the upper tokens which do not have a goal and move randomly, they do not record possible moves(next possible action) in the open list. Hence they do not have branches. For upper tokens which have goals, they need branches to record their possible moves(actions). In addition, paths(actions sequence) need to be recorded for all the upper tokens.
  Hence in the space complexity is slightly different from the one in Question 2 and the space complexity is $O(bL + mU + T) \approx O(m)$. <here we use b*L; L (which is # of lower tokens) should be similar to number of open list needed to expand node for all upper tokens that has goals >

Number of upper tokens < Number of lower tokens in the starting configuration:

- **Time complexity:** On average, for each upper token, they would have $\frac{U}{L}$ lower tokens to defeat. As the branching factor does not change, the time complexity will be O($b^{m*U/L}$).
- **Space Complexity:** The state dictionary and open_close_dict will renew after each movement, thus the space complexity will still be $O((b + m)U + T)$. Here depth m should be larger than the previous case.

Starting position of upper tokens and lower tokens:

- **Time complexity:** If upper tokens are very close to their target lower tokens, their search tree depth m will be very short, and thus less time will be required to find a solution.
  If upper tokens and corresponding targets are far from each other at the initial state, or there are many blocks that obstruct the movements of upper tokens, more time will be needed to find a solution.
- **Space Complexity:** There is less impact on the space complexity as state and open_close_dict dictionaries will keep renewing records. It's just depth m is larger if position is further from goal or a lot of blocks existes.