AI Project part A Report

Finally, you must briefly discuss your approach to solving this problem in a separate \_le called report.pdf (to be

submitted alongside your program). Your discussion should be structured around the following 3 questions:

* 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.)
  + Current game environment is:
    - Observable: can see all board and pieces at once
    - Deterministic: We know exactly what the board will be like once a move is performed
    - Static: The board state does not change until a move is made
    - Episodic: Each state can be solved independent of each other
    - Discrete: there is a large – yet finite number of series of moves to achieve a goal state or at all (however with a bad search algorithm it is possible to be stuck in a loop)
  + Each unique board, with number and position of each of its pieces, can be viewed as a unique note or state.
  + Goal is to move each piece we own off the board, or to take the “exit” action for all pieces at the requisite edge of the board
  + Actions: move, jump, exit. We want to move all the pieces off the board in as few moves as possible. Actually that might be more of a goal test.
  + Can be viewed as a search problem in two different states: (??? Is the first one really necessary tho?) 1. The board in its own physical representation of a search graph, with a limited amount of nodes, being each position. Some nodes are inaccessible even given an infinite number of moves, such as spaces with a “block” on them. 2. Initial state is one node. One of the pieces the AI controls is moved, new nodes are created with branching factor = sum of number of unique moves for each piece.
* 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.)
  + A\* is efficient because it explores nodes within an expected path cost boundary- nodes that (according to the heuristic) will obviously take longer path costs are not immediately explored
  + Eventually complete and optimal, even if we hit the resource limit and play is not optimal, it will be closer to optimality than BFS
  + Iterative depth first search was a consideration but almost certainly cannot complete the tree, thus is not optimal
  + Most efficient way for a piece to get off the map (best case scenario) is where pairs of pieces jump over each other continuously
  + Thus:
  + Heuristic:
  + *Code:*

*h = math.ceil(total\_distance/2) + minsum\_separation + num\_pieces*

* + - Total distance of pieces from edge of map, divided by 2 to assume each piece can jump. This value is ceilinged because if a piece is an odd number of unit distance from edge, it’s last move to get to edge cannot be a jump.
    - Minsum\_separation: In order for pairs of pieces to get to a position where they can jump over each other continuously to the edge of the map, the pieces must be next to each other with one piece facing the goal edge. We want to calculate the ‘separation’ between the pair of pieces’ current position and the position where the pieces are adjacent and ready to begin jumping continuously. If we project the pieces along the two non-goal axes, and identify the intersection closest to the goal edge, we obtain the most efficient position where the pieces can begin jumping. We don’t calculate the position, rather we directly calculate the number of steps it would take to get there. Piece 1 would move along one of the non-goal axes, and piece 2 would move along the other non-goal axis until the two pieces meet, minus 1 because one of the two pieces is adjacent to the intersection rather than directly on it  
      piece\_separation[p1][p2] = y\_dist + z\_dist – 1
    - Add number of pieces because each piece needs one turn to get off the map after reaching the edge
    - If there are an odd number of pieces, we assume the odd piece out can always jump (otherwise the heuristic is not admissible when there are blocks that the odd piece can jump over
* 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.) Your report can be written using
  + Each state to explore is a unique combination of pieces on the board
  + b - Branching factor = 4 pieces \* 6 directions possible to move = 24
    - Note that we used 6 directions rather than 12 possible moves (regular move or jump), this is because the possibility to move or jump is mutually exclusive (if there is a barrier, you can jump, otherwise you can move)
  + d – Goal Depth = minimum number of steps it takes to get all pieces off the board
    - This varies depending on the starting positions of pieces
    - Minimum of 4 (if pieces are already on the goal edge)
    - Maximum equal to max depth
    - Average? I don’t know
  + m – Maximum depth = combination of number of possible places for the pieces to be in
    - 37C4 = 66045 (37 possible squares on hex choose 4)
  + A\* in the worst case has O(bd) time complexity, where heuristic is 0 and it becomes a breadth first search
  + Time complexity improves dramatically with a good heuristic, thus most of the work has been on improving h(n)
    - An optimal heuristic would prune away all but one successor node, making the effective branching factor = 1, thus making the time complexity of the search O(n)
  + Space complexity in the worst case is O(bm), since each node explored needs to be stored in memory, but usually much lower because only the nodes contained by the largest explored f-contour is kept in memory
    - Note: f-contour is the set of all nodes whose f(n) is less than or equal (to the value of the contour), e.g. f-contour of 19 contains all nodes whose f(n) <= 19
  + The heuristic is the exact solution of a relaxed problem where pairs of pieces can always jump, and unpaired pieces can also always jump, and where there are only the player’s own pieces on the board
  + Thus the closer the program’s input matches this case, the more effective the heuristic will be (and thus lower time and space consumption, and lower effective branching factor)