**COMP30024 Artificial Intelligence**

**Project Part A:**

**Searching**

Team: James Hulonce (213977), John Whitham (xxx)

Due: 7/04/2020

The following provides a short report outlining our approach to solving the single player variant of Expendibots for part A of the assignment.

**The game formulated as a search problem**

The first step we took was to break the problem down into its search problem elements, including: states, actions, goal tests, and path costs. The following outlines how we treated each of these key elements in our solutions:

* **States:** We assumed that the main state relevant for this problem is the board setup. That is the location of each piece on the board. E.G. {'white': [[3, 1, 1],[3,4,4]],'black': [[1, 3, 3]]}.
  + Note: the board setup recorded each piece as a stack, and recorded its colour, stack size n, and coordinates x and y.
* **Actions**: Our assumed action space included, for each white stack, moving up to n pieces (n is the size of the stack) up to n spaces left, right, up or down; and exploding the stack.
  + Note that the action space was restricted by what are legal moves. Illegal moves included: moving pieces off the board, and moving white pieces onto black piece.
* **Goal tests**: The goal test we used what a state where there were no black pieces remaining. The rules noted that, even if all the white pieces are destroyed, it still counts as a win if all black pieces are removed. As such, the goal state is simply all black pieces are exploded.
* **Path costs**: our algorithm does not explicitly use a path cost, but it does use a heuristic which is outlined below.

**Choice of search algorithm, and why**

For the next step we considered a number of algorithms, and choose the one we believed was most appropriate. The key algorithms we considered included: breadth first search, depth first search, greedy best first search and A\* search.

As the branching factor of this problem can be quite high and the search space can quickly get quite large, we initially discarded both breadth first search and depth first search as they were likely only able to answer simple test case. Also, we thought that this problem could make use of a heuristic function.

So, from here we decided to use an algorithm based largely on greedy best first search algorithm using a custom heuristic to decide which nodes to expand next. We choose this algorithm because we believed for this problem the history of getting to a state was not overly important, nor was the cost to get there.

The **heuristic function** we used was one which measures the Manhattan distance between each white piece and each black piece, then subtracted the distance between the white pieces. Our thought process was that we wanted to spread out the white pieces (to ensure all the black pieces get removed when exploded) while also moving the white pieces towards black pieces to explode them.

The heuristic still allows for stacks to form to jump over pieces as needed, however it looks at many options before that.

This heuristic is unlikely to be **admissible** as it does not closely represent the cost of reaching the goal. It is more a method of directing the search in a way that we believe will cut down the search time.

The efficiency, completeness and optimality of our solution are discussed below:

* **Efficiency** – the space and time complexity of this algorithm is generally O(bm), where b is the branching factor and m is the maximum depth of the search tree. In the worst case the algorithm will look at all bm nodes and needs to hold them all in memory.
* **Completeness** – greedy search in general is not complete, as it can get caught in loops. It may be possible for it to get stuck in loops for this specific problem, however our testing so far has not found any such cases. One way to improve this and ensure completeness would be to add a cost of an action/ cost of moves to a given state i.e. A\*. But as this has worked we did not implement this additional functionality as it did not seem needed.
* **Optimality** – the algorithm we implemented does not record the actions taken to get to a specific node (like A\*) and as such it does not have a mechanism to ensure optimality. Our algorithm is not optimal.

**Features of the problem and your program's input impact your program's time and space requirement.**

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

A key factor which impacts the time requirement of our algorithm is the number of white pieces on the board. As the number of white pieces increase, the branching factor of the algorithm increases which leads to a higher time complexity, noting O(bm).

Similarly, as the difficulty of the problem increases (i.e. the number of pieces or the need for pieces to jump to solve the problem) the time requirement will also increase. This is largely because our heuristic does not encourage stacks, and so if a stack is needed it will need to process many nodes before a solution is found.

[any other thoughts??]