COMP30024 Artificial Intelligence

Project Part A: Searching

Team: RageAgainstTheSentientMachine

James Hulonce (213977), John Whitham (1023660)

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.
    - Note: the board setup recorded each piece as a stack, and recorded its colour, stack size n, and coordinates x and y. E.G. {'white': {(x, y): n, (4, 4): 3}, 'black': {(3, 3): 1}}.
- **Actions**: Our assumed action space included, for each white stack, moving 1 to n tokens (n is the size of the stack) up to n spaces left, right, up or down; and exploding the stack.
    - For example a stack of 2 tokens with nothing blocking it would have 17 possible actions. These actions would include (for each of the cardinal directions) move one token one space, one token two spaces, moving two tokens one space, moving two tokens two spaces; and exploding.
    - Note that the action space was restricted by what are legal moves. Illegal moves included: moving stacks off the board, moving white stacks onto a black stack and not moving the white stack.
- **Goal tests**: The goal test we used what was 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 high (for example with stacks) and therefore the search space can quickly get large, we initially discarded both breadth first search and depth first search as they were likely only able to answer simple test cases.

We then look further at those algorithms which use a heuristic function. This was primarily because we could think of a few heuristics which could help improve the algorithm which use the distance between stacks.

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.

chose

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 non-stack options first (which is relatively quick). This approach also keeps the branching factor low initially (few stacks), unless a solution is not found and then moves to create more stacks. This helped optimise the speed of our algorithm at solving the test cases.

Could also say spread white out to minimise friendly fire

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 **$O(b^m)$,** 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 b^m nodes and need to hold them all in memory.
- **Completeness** – greedy search in general is not complete (i.e. it is not guaranteed to find a solution if one exists), this is because it can get caught in loops. To counter this, our algorithm stores previously visited nodes, checks new nodes against this list and doesn't add it to the queue if we have visited this node before. Additionally, because of this our search space is finite, given this specific problem our algorithm **is complete**, and will find a solution if one exists.
- **Optimality** – the algorithm we implemented does not record the cost to get to a specific node (like A*) and does not have a mechanism to ensure optimality. Our algorithm is not optimal.
    - One way to improve this, and further ensure optimality would be to add a cost of an action/ cost of moves to a given state i.e. A*. However, our algorithm worked for all test cases and our own custom more difficult cases, as such we did not implement this additional functionality as it did not seem needed.

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

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 $O(b^m)$.

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 (of more than 1 token), and so if a stack is needed it will need to process many nodes before a solution is found.

As noted earlier, when stacks are formed the branching factor increases exponentially. As such, any problem requiring stacks to find a solution will increase our programs space and time requirements as more nodes will need to be visited and stored.

I found with the custom test cases that the worst case was when n_white = 3 and n_black = 2. The 'attraction force' from the black tokens was weaker than the 'repulsion force' from the white tokens. Having to make stacks doesn't appear to make the algorithm suffer