COMP AI Sem 1 2020 Part A Project Report

Written by Ash Knipler (999 159) and Girish Madnani (934 130)

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.)

We approached the problem by identifying our actions and goal test.
Our actions were limited to moving in four directions or exploding. These rules are not hard to implement in code, with the four directions being implemented by using a Manhattan Distance function and move_coordinates scheme.
The goal test was if all the black tiles had exploded. We solved this by finding our solutions first and knowing that if all our white tiles were in those solutions spaces, then their explosions would solve the problem.
We viewed the problem as having two stages. First, find where the white tiles need to be in order to solve the problem. Second, find the path to get the white tiles to their target location. Doing it this way could theoretically save a lot of time and space complexity for our program.

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.)

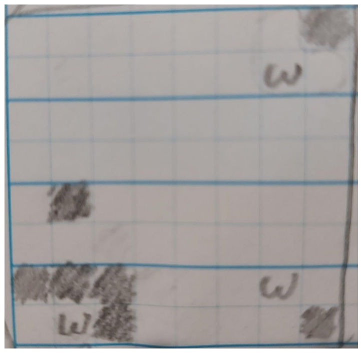
Our program conducts two searches. The first is to find the coordinates required by the white coins to solve the problem. To reduce the space and time requirements the process is simplified to the utmost. We created a find solution () that enabled us to find all the possible positions of the white coins in order to explode and get the solution. This search problem uses a Dijkstra algorithm that runs through all the possibilities of the simplified possible solution set.
Secondly, we ran an A* search algorithm with a heuristic of Manhattan Distance to find the path to these locations, solving requirements such as if there was a blockage by the black coins.
The A* search algorithm was chosen as it is an advanced search that is comparative and saves time. It can be optimal and complete with the right heuristic but failing that it has a high chance of being faster than most search algorithms, such as breadth-first search.
A* is a combination of Dijkstra's algorithm and Greedy Best-First-Search algorithm that usually gives ideal results.

COMP AI Sem 1 2020 Part A Project Report

Written by Ash Knipler (999 159) and Girish Madnani (934 130)

We used Manhattan Distance rather than Chebyshev Distance as we do not have to move diagonally. It is also preferable over Euclidean Distance as we do not have weights and the whole board is the coordinate system.

The completeness of our program is in direct line with the capabilities of our move function. Currently it solves all test cases, although we are aware that it would not solve a case where a single white tile is trapped and must be 'rescued' to complete the problem (such as shown). Our move function could be improved to accommodate this case, by modifying 2 or 3 factors and implementing 2 more functions, however we did not have the time to put this together. As such our current program is incomplete and not optimal but it does solve for all test cases and is not far from being complete. Our heuristic is faster as it doesn't have as many factors as a heuristic which was also calculating for the BOOM mechanic.



<u>What features of the problem and your program's input impact your program's time and space requirements?</u>

(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.)

The complexity of the white tiles path directly impacts our programs time and space requirements. Our program is slow to find solutions that require "jumping" over black tiles to solve the problem. The pathfinding is definitely the slowest part of our program, in particular having an impact of potentially $O(2^d)$ where d=the largest stack a white tile must become. This is because the branching factor of the search tree increases by 4 when a stack's height increases by 1.

Overall the time complexity for our A* algorithm is around $O(b^d)$. This is because our heuristic would only have a coefficient reduction.

In summary, the program does not become significantly slower when more black tiles are introduced, instead becoming significantly slower when more white tiles are introduced, or more complex pathfinding is required.