COMP30024

Artificial Intelligence

Part A - Report

Deevesh Shanmuganathan - 912364

Jason Polychronopoulos - 921565

Formulation:

The single player variant of Expendibots is subject to the following environments:

- *Fully observable*: All pieces on the board are known and measurable.
- *Deterministic*: All outcomes of actions are certain.
- *Sequential*: Memory of past actions is needed to determine the next best action.
- *Static*: The environment is not constantly changing, the state remains the same while actions are being decided.
- *Discrete*: There are a finite number of actions and outcomes.

Formulation of the simplified version of the game as a search problem was accomplished by representing each game state as a node in a tree, where traversal and expansion would occur by applying discrete moves as actions to a singular stack . All possible actions include moving a number of pieces in a stack left, right, up and down by a finite number of steps.

Goal states were created by grouping black stacks that would be impacted by a chain explosion. Testing of goal states involved finding a state where an elected white stack could blow up an assigned black stack or group of black stacks while attempting to minimise collateral damage amongst the white stacks. The game ends when there are no more black stacks on the board, which implicitly happens as a result of each previous goal state being reached by gradually blowing up groups of black stacks.

The path cost involved aggregating the total number of movement actions made, which would be equivalent to finding the depth of the search tree to the solution.

Algorithm:

Our algorithm of choice was Iterative Deepening A* Search (IDA*), as we found it reduced the search space considerably compared to Iterative Deepening Depth First Search. We also considered A* but chose to implement IDA* as it requires less memory. States would be evaluated using the relation f(n) = g(n) + h(n), where g(n) is the total path cost to reach the current state, and h(n) is a heuristic value to estimate the cost of reaching the next state.

The benefit of using an A* related search algorithm is that with an admissible heuristic, the search is optimal. We used manhattan distance as our admissible heuristic, which is always less than or equal to the true path cost of getting a white stack to a group of black stacks as it is a relaxed version of the true path cost calculation. Therefore our algorithm is optimal.

A heuristic ensures that our algorithm is optimally efficient in finding the solution as it is guided by a combination of path costs and heuristic values. This allows a solution to be found faster when compared to algorithms such as Uniform Cost Search which only take into account the next lowest path cost, which could result in an overall higher cost path.

Furthermore, the algorithm is complete given a solution exists, which for the purpose of this task is guaranteed. This is because the algorithm explores all paths eventually as the threshold iteratively increases, meaning that the solution will always eventually be found if it is in the search space.

Time and space requirements:

Our team elected to focus on moving one white stack at a time, moving it towards an assigned location, whether that be a single black stack or a position where multiple black stacks could be blown up. Focusing on one stack at a time enabled us to reduce the branching factor of each tree, where a stack could have a maximum of 4*N*N possible moves, where N is the number of pieces in a stack and also the number of possible steps that can be taken by a stack. Although, we would have to iterate search for every white stack on the board.

The input for this program has many dimensions to process. Each stack has 4*N*N branches, as discussed above. Multiplied by the total number of stacks, the branching factor can overwhelm the search space of the problem. In order to avoid this, we broke down the problem by solving one group of black stacks at a time and only moving the single closest white stack. By reducing the branching factor, we overall reduce the space and time complexity of our search algorithm.

Compared to A*, IDA* has no need to store all nodes in memory as the algorithm only searches up to a certain value of f(n) before restarting the search with an updated threshold. As a result, we were able to achieve a space complexity of O(d), where d is the maximal depth of the tree. This is because it only needs to store knowledge of nodes on the current path until a threshold or solution node is reached. A* on the other hand has a space complexity of O(b^d) as all nodes need to be kept in memory to track path costs.

However, IDA* requires more processing power as it has to explore the same nodes multiple times on different iterations. This additively results in a higher Time complexity than the A* algorithm, despite both algorithms having the same asymptotic time complexity of (b^m), where b = branching factor and m = maximal depth. Given that we were allowed a generous amount of time (30 seconds) and a limited amount of space to run our solution, our team decided to implement IDA* as it optimised our constraints.