Jiles Denham [914740] and Nima Karimi [995785]

Comp30024 Artificial Intelligence Part A Analysis

In terms of search environment, we have formulated the environment presented for Part A as fully observable, single-agent, deterministic, static, discrete, and completely known. The only changes in board configuration come from actions taken by the search algorithm moving a piece, and choosing to boom once the pieces are in place, with the agent being able to predict the full outcome of any taken action due to the deterministic nature of the environment. Due to this fully observable and deterministic nature, if suitable goal(s) were formulated, a good heuristic would be able to guide the search algorithm straight towards the goal(s) as the algorithm knows the exact outcome of the actions it takes. The approach we took for solving the problem involved two key steps, first searching for the goal coordinate(s) that would result in a solution if a white piece was boomed, and then finding the paths the white piece(s) would take to those goal coordinate(s) from the initial board configuration.

There are two separate goal formulation algorithms that are present, one for stage 2 problems the other for stage 3, however both are very similar algorithmically. This approach for goal analysis was favoured over others as it has little time and space requirements and has similar logic an intelligent being who would be analysing their goal while being accurate with little chance of error. The algorithm revolves around the idea of searching through every single tile and running a test explosion on that tile with testExplode(), which imitates what would happen if a white piece were to explode on the tile; hence recursively exploding all the pieces that are in the chain explosion, finally returning a two element list consisting of the starting position and a list of the pieces caught in the explosion. Within testExplode(), there is a loop that checks 8 tiles around a position (still iterates 8 times even if position is on edge or corner) and can then call testExplode() again depending on whether there is a piece on the position or not. This means that testExplode() will be run 64 times (for each tile), which then checks 8 tiles a max of b times, where b is the number of black pieces. Hence resulting in a maximum of 64 * 8 * b iterations, where 2 <= b <= 12.

For stage 2, the algorithm obtains the list of explosions and removes any duplicates (i.e. the explosions that consist of the exact same set of pieces that were seen in a previous explosion), then finds the explosion which the highest amount of black pieces caught, as long as the number of white pieces caught in the explosion are explicitly less than or equal to the number black pieces caught in the explosion. For checking for duplicate explosions, the array of pieces needs to be sorted; this results in n * log(n) time for the sort, where n is the number of pieces within the explosion array and 0 <= n <= b. Furthermore, there is a final for loop which checks every value within every explosion array to determine a final goal position which uses a time of k * n, where k is the number of unique explosions that are found. In the case where we only have 1 piece that can explode all of the black pieces, we can assume that 1 <= k <= 2. Assuming worst case and n = b for all cases, this would result in 512 * b + b*log(b) + 2b = 514*b + b*log(b); hence, O(n * log (n)) time complexity (even though 514 * b > b * log (b) for 2 <= b <= 12).

Stage 3 follows a similar structure but instead of removing all duplicate explosions, it first finds the biggest explosion and adds all the positions that could cause that explosion to an array of *n* size (where *n* is equal to the number of white pieces on the board) in the first element. The algorithm then assumes that all the pieces in biggest explosion are now gone, and proceeds to find the next biggest explosion that includes any of the remaining pieces and repeats the process for *n* times. This leaves a list with the format *goals = [[goal_list_1], …, [goal_list_n]]*, which the search algorithm then uses to determine the best move for the white pieces to make such that piece 1 can move to any position in goal_list_1, piece 2 can move to any position in goal_list_2 and so on. This algorithm uses the same sections from stage 2 which results in 514 * b + b*log(b).In the findGoalArray() function, we are reusing code from stage 2 but are also adding all exploded pieces uniquely to an array based on the current biggest explosion, which uses i * j where i is the number of already exploded pieces and j is the number of pieces in the current biggest explosion while 0 <= i <= b and 0 <= j <= b. The algorithm also checks all positions in all explosions to see whether the explosion covers any of the remaining pieces which takes k * z * n time, where z = n – i, k in this example is the number of unique explosions and can be assumed 1 <= k <= 6 (as only 6 clusters can exist for 3 white pieces). Assuming worst case scenario and that i,j,z,n = b, this gives a time complexity of b^2 + 6 * b^2 + 514 * b + b*log(b) = 7*b^2 + 514*b + b*log(b), which results in O(n^2) time.

The search algorithm, a_star(), is then implemented to find the shortest path a white piece must travel to reach the goal. If multiple goals had been formulated, in the case of solving stage 3 problems, then a single white piece is delegated a suitable goal using goal_select() with selection based on the lowest sum of Manhattan Distances of each white piece to each goal. Each of these respective white piece/goal pairs are then passed to a_star().

For the purposes of search, the internal state of the search algorithm is only concerned with the single white piece that it has been given. Since stacking and stage 4 is not implemented in this program, there is no need to account for all white pieces when finding the path from any white piece to its delegated goal. The configuration of black pieces is also passed as an argument, though these are treated as an unchanging aspect of the environment and are not included in state representation as they do not change with actions undertaken by the search algorithm. The representation of actions for the construction of the search tree used by a_star() is represented by the class StateNode, with the class attributes being state, path cost, heuristic value, and parent. State contains a single coordinate of a white piece, path cost the number of moves from the initial state to the current state (also can be seen as depth of the search tree) with a uniform increase of 1 per action undertaken, heuristic as the Manhattan Distance of the current state to the goal, and parent as the StateNode from which the current StateNode was generated.

A_star() was chosen for its 'greedy' tendencies based on both path cost and heuristic value, added together to form an evaluation function. However, when choosing which StateNode to eject from the priority queue, if the evaluation function of x StateNodes is equal, then priority is given to the one with a lower heuristic (expectedly closer to the goal). Since the search environment is fully observable and deterministic, we felt it was the right choice to give a lot of priority to the Manhattan Distance heuristic as an accurate evaluation of how close a white piece was to the goal. Due to this our search algorithm behaves a lot like greedy best-first search, but with a record of explored StateNodes to stop infinite backtracking and ensure completeness. For the vast majority of inputs, our a_star() tends to search deeply into the search tree instead of wide, going straight for the goal.

To improve time complexity, our implementation of a_star does not check whether a node is in already in frontier but with a lower path cost. Because of this omission, our a_star() can't be guaranteed to be optimal for all cases, however the Manhattan Distance heuristic is both admissible and consistent for this search problem (this would not be the case if stacking was implemented). For the majority of cases our a_star() is both optimal and complete, with optimality only being jeopardised with inputs where a white piece is boxed into a corner and has to go around a wall of black pieces. However, in terms of the overall solution of the problem, the search is far from optimal as the goal(s) given as input have no guarantee of being the closest to the white piece in Manhattan Distance, and thus the algorithm only fulfills the above criteria in relation to the goal it has been given as input.

StateNode.successors() represents the transition model/potential moves that may be made, resulting in a usual branching factor of 3 (4 cardinal directions take 1 for the coordinate it came from) for moves that are not obstructed by the edge of the board or a black piece. We considered this a modest increase in space complexity in relation to goal depth and thus didn't feel the need to implement something like simple memory-bounded A-star. However, space complexity is still exponential (O(b^d)) due to keeping all generated StateNodes in memory. For time complexity, the accuracy of the heuristic usually results in a very low relative error which is equal to (actual cost – heuristic value)/actual cost. Since step-costs in the search space are uniform, we can define the time complexity as O(b^(re*d)), where b = branching factor, re = relative error, and d = solution depth. Judging from this, inputs with high solution depth will result in exponentially more time and space complexity for our search, along with inputs that obstruct the Manhattan Distance path of white piece to goal.