COMP30024 Artificial Intelligence: Project Part A

Our agent aims to find a smallest number of legal game moves which will move all of a given player's pieces off their corresponding 'goal edge' on the Chexer's board.

1 Chexers as a Search Problem

To treat this as an abstract search problem, we treat any legal configuration of the board as a *state* - i.e. coordinates of the player pieces and blocks - and the legal game moves a player can make as transitionary *actions* between states. To check if a given state is a goal state, we simply verify that none of the player's pieces are left on the board. Since the required solution was a minimal number of moves to the goal state, without any preference on the type of moves, we assign a cost of 1 to any single action.
In terms of the actual implementatin, we only store the coordinates of the non-exited pieces in the state, as the block positions do not change from state to state. We store all other information externally as a part of the "hexboard" to reduce information redundancy. When expanding a node, we refer to our hexboard to determine legal actions.

2 Search Algorithm choice

We opted to for the A* search algorithm, as the best path is highly sensitive to the configuration of the board, and an informed search algorithm would take advantage of this information to reduce and guide it's search spoace. It is also guaranteed to be optimal provided the heuristic is admissible. In this case, A* search is certainly complete, as it will only be incomplete when there are infinitely many nodes with a $f < f(G)$, where f is the evaluation function, and G is an optimal solution. Since we are checking for repeated states, there are only finitely many states available on the finite board, and hence there are only finitely many possible nodes in our search space.

2.1 Heuristic Calculation

Initially our heuristic was simply a result of a calculation of distance from the exit tiles (the tiles from which the player pieces can exit). We used the formula $heuristic = Distance/2 + 1$ to associate a heuristic value to each tile (occupied or not). The actual heuristic value that was associated with a given state was the sum of the heuristic values of each tile that was occupied by a player piece. The division in the formula was used in order to keep the heuristic admissible, since player pieces may make a jump action in the right condition, covering two tiles with a cost of 1. The '+1' is to allow for the exit cost of a piece. This heuristic is clearly admissible, as it calculates the exact cost of a solution from a given state where a piece can jump as often as it likes over empty tiles, unhindered by any obstacles. The actual solution to the problem must always have an equal or greater cost, since the pieces may be hindered by obstacles or an inability to jump. We originally kept this formula as a float so that tiles closer to the exit always held a lower heuristic, even though the lowest number of moves to exit (assuming maximum number of jumps) does not always increase as you move closer to the exit (as every second row requires a non-jump action at some stage to land on an exit tile). However, when we changed our heuristic formula to $heuristic = \lceil Distance/2 \rceil + 1$, our agent appeared to perform much more efficiently for many test cases in a row. Before we could investigate further, we came up with a new heuristic calculation that far surpassed the performance of the above heuristics.

2.2 Dijkstra Based Heuristic

We decided to try a heuristic that took into account the placement of the block tiles on the board. Each tile was given a heuristic value representing the fastest way to reach that tile from an exit tile, allowing for the block tiles. This was achieved by running Dijkstra's algorithm with the tiles of the board as the vertices of the search graph, and moves/jumps as cost-1 edges between tiles, beginning with the exit tiles set to a cost of 1 (to account for exit move). This finds the shortest path between the exit tiles and any other tile on the board. In order to keep this heuristic admissible, during its calculation, we allowed jump moves over empty squares (as long as it lands on an empty tile), as it is possible that another player piece could be used to achieve this jump in practice. Now when calculating the heuristic of the state, we simply total the precomputed values of the tile heuristics at each player piece. This total value represents the exact smallest number of moves/jumps it would take all of the pieces to exit the board, if they were allowed to jump over empty tiles, and didn't care about landing on each other. Hence it must be admissible, but is certainly much more informed. Hence there is less error between the true cost and evaluation cost of each node, making the search more efficient. This heuristic is maximally helpful when the direct path to the goal is blocked, as the heuristic will guide the search towards the correct path.

3 Strains on Time and Space Complexity

We came across 2 major influences on the overall complexity of the search algorithm

3.1 Branching Factor

The branching factor of this search problem likely had the greatest . In the case of finding the shortest path for a single piece, there can be up to 6 possible actions, each a move, jump or exit action. This can be quite easily managed at the small scale of this board. However, this branching factor increases to a maximum of 24 when there are 4 pieces on the board simultaneously. This branching factor would prove disastrous under an uninformed search algorithm, as we would be exploring paths that take the pieces in the "wrong" direction with the same priority as those heading the "right" way. The heuristic used in A* provides this sense of direction, which greatly reduces the explosiveness of the branching factor.

3.2 Revisting States

Due to the massive branching factor, there are many, many ways to reach a particular state via distinct sequences of moves. Often a sequences of moves can be performed in a different order, still resulting in the same state. This becomes especially apparent on boards with minimal blocks and maximal player pieces arranged far apart from each other. There is almost no restriction on the possible moves to choose from, resulting in thousands of repeated node expansions. By storing a set of these states already "seen", we were able to dramatically reduce the search space of our algorithm.

3.3 Other comments on complexity

Due to our use of Dijkstra's algorithm to precompute heuristic values for each tile, our algorithm actually performs equally well on inputs with counterintuitive paths to the goal (which begin by moving away from the goal). In general, we found that the more the blocks, the less the action options and the smaller the search space.