Artificial Intelligence Project Part A Report
Danielle Zhang & Ishan Sodhi

The game Chexers has been formulated as a search problem by taking into consideration the following components:

State: The state refers to the positions of all pieces on the board.

Actions: The possible actions for the pieces have been defined as “MOVE”, “JUMP” and “EXIT”:

- “MOVE” - The pieces move from one position to a neighbouring available position. When the move action is implemented either or both the row and column changes by +1 or -1.

- “JUMP” - When the immediate neighbour of any piece is blocked by another block or player, the JUMP move is initiated, either one of the row and column coordinate or both the row coordinate and column coordinate change by +2, -2 .

- “EXIT” - When a piece reaches its predefined exit coordinates or position, the “EXIT” move is initiated and the coordinates of the piece is set to the “removed” coordinate (-5,-5), which reflects that the piece is no longer on the board.

Goal test: The goal test check whether all the pieces have exited the board, when the value of all the players on the board becomes (-5,-5). When the goal test is successful the game should become over.

Path cost: The cost is 1 when a piece moves from one position on the board to its neighbouring position, or jumps over an obstacle or other piece to reach another position. The total path cost would be the total cost for moving all the pieces from their initial positions to the “removed” coordinates.

The search algorithm used to solve this problem is the A* algorithm, with pre-computed shortest paths using Dijkstra’s algorithm as the heuristic.

A* is a well balanced algorithm in the sense that it combines both the shortest distance from the Dijkstra algorithm with heuristic search to prune the nodes most effectively. It chooses nodes to visit according to the sum of the the lowest path cost and the heuristic value to the goal (the value estimated closest to the true value). This allows it to prune out nodes that are both in a direction farther from the goal and from out of these, choose the nodes with the shortest path cost.

In order to have a heuristic function that is close to the truth, Dijkstra’s algorithm is run at the start. We use Dijkstra’s algorithm to find the shortest path from every position that is not occupied by the blocks, to any exit positions that are not occupied by the blocks, with jump actions considered. These shortest paths are stored in a dictionary and can be accessed when calculating the heuristic for each state. Every position’s shortest path to the exit position is also the shortest path for a piece on that position to the exit when there are no other pieces on the board. However, when there is more than one piece on the board, the optimal strategy for

shortest solution is to make these pieces close to each other and use each other to perform jump action. The true shortest path for every piece to the exits will be shorter than the Dijkstra's shortest path. Therefore in our heuristic evaluation for every state, we sum the Dijkstra's shortest path of every piece that is still on the board, and then multiply the sum by ¾. This heuristic is not guaranteed to be admissible, but the approximated cost it produces at the start of the solution search is not only smaller but also very close to the real cost when the program is run on various test cases.

In terms of the trade-off between time and space complexity, it will visit a balanced number of nodes in a relatively short amount of time ultimately allowing us to find the shortest path most effectively.

Since the number of positions on the board doesn't change, running Dijkstra's algorithm and storing the shortest paths Dijkstra's algorithm found at the start of the search have time complexity O(1) and space complexity O(1). Hence A* algorithm impact the program's time and space complexity the most. The depth of the search tree d is the length of the shortest solution. Since each coordinate has maximum 6 neighbours, in A*, if n pieces are given, the maximum branching factor b is 6n. The worst time complexity of A* is $O(b^d)$. And since A* keeps every node in memory, the space complexity is also $O(b^d)$. However, A*'s performance is highly dependent on heuristic function's accuracy. At any level of the search tree, if the heuristic function give i nodes incorrect estimations that are lower than the node with the lowest true cost, then these i nodes are expanded first before A* expands the node that actually has the lowest true cost. Hence the actual maximum branching factor b* is the maximum number of nodes that the heuristic function wrongly estimated to be lower than the lowest true cost. Since our heuristic function is sensible, b* is smaller than b. Hence the true time complexity $O(b*^d)$ grows slower than $O(b^d)$.

Worst case behaviour : The worst case behaviour for any algorithm would be observed when there is a chunk of blocks blocking the pieces to the exit positions. Other heuristics such as Euclidean distance, Axial distance or Manhattan distance would underestimate the heuristic value as they do not know if there are any blocks in the path, leading to the algorithm exploring a lot more nodes than required and wasting time as well.

As our algorithm uses Dijkstra distance as the Heuristic distance, it already takes into account if any chunk of blocks are blocking the pieces to the exit position, and calculates the heuristic distances that walk around the blocks, which gives us a measure much closer to the true value, allowing us to prune out a lot more nodes more effectively. Our algorithm also takes into account if any of the pieces can jump over other pieces to reach the goal more quickly and gives us a very accurate heuristic value.