COMP30024 - Project A Report

Akira Wang (913391), Callum Holmes (899251)

PROBLEM FORMULATION

The problem of navigating pieces across a Chexers board to a successful exit is:

1. Fully Observable: All exits are known, and all piece coordinates are measurable.
2. Deterministic: there is no stochastic factor in the problem, all outcomes of actions are certain.
3. Sequential: The benefit of a given move.
4. Static: There is no time factor in the actual problem, the game state/environment remains equal whilst actions are being decided.
5. Discrete: There are a finite number of outcomes, all pieces are exited or not.

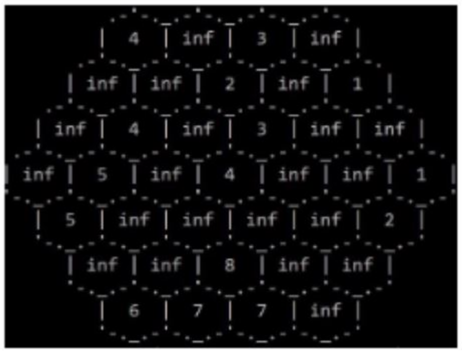
The problem was interpreted as a search problem on a directed graph of possible game states, reachable by player actions. States are defined by the exact number and positions of each player piece and block. Actions possible are to move or jump left, right or diagonally (where space is available) or exit if located on the correct hexes - and these are uniformly costed. The goal test is to check if no more pieces are on the board - if and only if this is true, has the goal been reached. Lastly, path costs are calculated as the total number of actions made, and so an optimal path is one that reaches the goal with the least possible cost from the initial state.

ALGORITHMIC APPROACH

Retrograde Dijkstra Heuristic

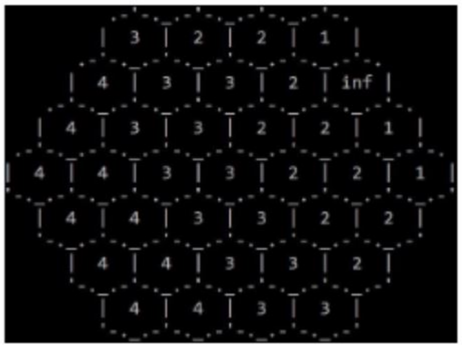
Our heuristic evaluates the total minimal cost to exit all pieces if they move independently under relaxed conditions. This relaxed condition allowed piece(s) to make legal jumps whenever possible without the condition of a block or piece in front of the direction of jump. Hence, pieces would not need to converge in attempt to enter a "leapfrog" pattern of moves in order to reduce path costs, as pieces pieces could reach the goal independently at minimum cost. The Retrograde Dijkstra Heuristic preprocesses the board state with blocks (excluding player pieces), using an adapted Dijkstra graph search algorithm to map the minimum cost to exit for a piece at any location. To evaluate a state, the cost associated with each piece position is aggregated.

Figure 1. Example Cost Evaluations



```
      |  4  | inf |  3  | inf |
   | inf | inf |  2  | inf |  1  |
| inf |  4  | inf |  3  | inf | inf |
| inf |  5  | inf |  4  | inf | inf |  1  |
|  5  | inf | inf | inf | inf |  2  |
   | inf | inf |  8  | inf | inf |
      |  6  |  7  |  7  | inf |
```

(a) Dense state



```
      |  3  |  2  |  2  |  1  |
   |  4  |  3  |  3  |  2  | inf |
|  4  |  3  |  3  |  2  |  2  |  1  |
|  4  |  4  |  3  |  3  |  2  |  2  |  1  |
|  4  |  4  |  3  |  3  |  2  |  2  |
   |  4  |  4  |  3  |  3  |  2  |
      |  4  |  4  |  3  |  3  |
```

(b) Sparse state

Analysis of the Retrograde Dijkstra Heuristic

1. Benefits
   (a) Admissibility: The heuristic is admissible. The only difference between the relaxed and actual version of the problem is that 'relaxed pieces' can jump more often - meaning that they will never reach the goal slower than pieces in the actual problem.
   (b) Partly memoizable: Only the blocks and goal positions determine the cost evaluation for each hex, which are static in this problem. Hence the board costs only need computing once; then the heuristic can be evaluated in constant $O(m$ time/space computation, m being the number of player pieces.
2. Limitations

   For dense boards this heuristic can find optimal jumping paths and weight them as the best action to take - looking at a dense state in **Figure 1 (a)**, pieces on this board would have ample jumping opportunities even if isolated. However, for sparse graphs with few blocks, the heuristic degenerates to uniform-cost search. Looking at a sparse state **Figure 2 (b)**, pieces on the bottom-left fringe would be indifferent to moving sideways or forwards.

Furthermore, as jumping is always possible in the relaxed version, the relaxed problem does not care if pieces try to jump over each other. Consequently the heuristics fails to value movement that allows 'leapfrogging' (where a pair of pieces jump over each other repeatedly), which requires pieces make moves that converge towards other pieces. As leapfrogging is a stronger option on sparse boards, overall the heuristic significantly underestimates true solution cost in sparse graphs due to its simplifying assumptions.

IDA Search Algorithm*

Iterative Deepening A* (IDA*) was used to search the game tree. IDA* uses the best-first search approach of A*, evaluating states as $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost to reach the state from the initial state, and $h(n)$ a heuristic estimate of the cost to reach the goal from the state. Both algorithms are *optimal* given an admissible heuristic is used and are *complete* if solution states exist (which is assumed for this problem).[1]

We found in experimentation that IDA* had stronger performance on average compared to A* even with optimisations, and so was our algorithm of choice.

PROBLEM AND INPUT COMPLEXITY AND OPTIMISATIONS

Program Input Considerations

Since the program input only contained a string representation for the player and two lists for the pieces and blocks, it was incomplete. Coordinates of blocks and pieces are never altered, the primary use of a state is to evaluate membership of other locations (to derive potential actions). Hence possible exit locations (player goals) and valid board coordinates (in axial coordinates) were pre-defined as global variables to avoid recalculations.

Program Considerations

1. Depth: The depth of the game tree was primarily affected by the initial state and the number of player pieces. This is because some initial states varied in path cost to achieve goal state from a range of 1 to over 30 steps. Due to the nature of potential actions, a state could be entered several times by backtracking, adding more depth and diluting the game tree with repetitions.

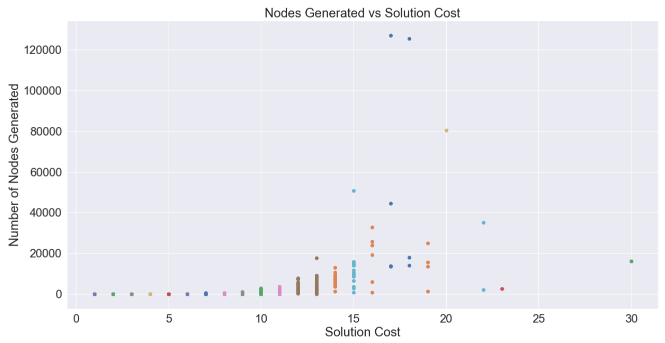
2. Branching: In the worst case scenario, the state would have four sparsely spread pieces, resulting in a game state having up to 24 possible actions (6 possible actions per piece). As a result, the game tree would potentially become highly dense, which encroaches on time and space complexity.

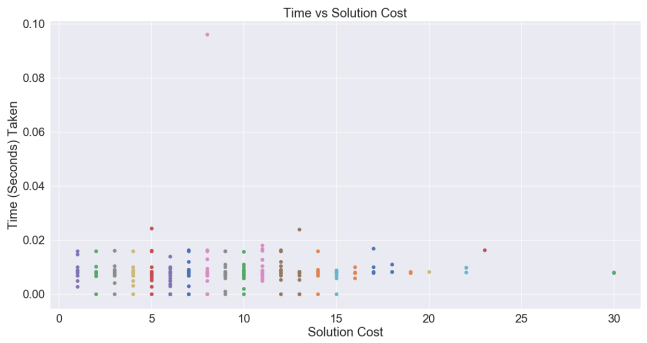
Optimisations

1. Exit First Optimisation: The algorithm was programmed to prioritise exit actions at any stage. This reduced branching factor towards the final stages of the problem.

2. Transposition Table: A transposition table eliminated repetition down and across the branches. This ensured that every state was expanded at most once where the most optimal instance for every state expanded so far was considered, trimming nodes whose evaluation was not better than a previous encounter of the same state. The transposition table was indexed using a custom zero-collision hash function (since the in-built Python hash function had several collisions).

TIME-SPACE COMPLEXITY ANALYSIS





Nodes generated follow an $O(b^d)$ space complexity, which is reasonable as the algorithm stores a copy of each unique state. Time is approximately $O(n)$ given results, albeit it would be exponentially complex for hard cases.

[1] https://ai.stackexchange.com/questions/8821/how-is-iterative-deepening-a-better-than-a