2802ICT Intelligent Systems
Assignment 1 Milestone 2 Report

By Nathan Cowan
S5143344

Problem Formulation:
As explained in the previous report the board is arranged in a 6x6 grid (represented as a string or a list of characters) in which there are line pieces labeled A-K if they are 2 long or O-R if they are 3 long. There is also the special piece X which is always 2 long and placed horizontally somewhere on the 3rd row. Any piece can only move along its axis.

The possible state space itself is vast and not trivially calculable, however an upper bound can be given as 16^36. That is the number of possible pieces to the power of the number of board spaces. As this is only an upper bound the number of valid/reachable board states are significantly lower than this.

The Initial State in this case is any of the 40 problems given in ‘rh.txt’ but more generally can be any valid state.

The only possible action is to move a piece some number of spaces in one direction or another along its axis.

The state transition is almost as simple except that attempting to move into a space that is already taken will result in nothing happening. This is filtered out automatically in the expand function.

The goal state is any state where the right-most space of the 3rd row is ‘X’ or the 18th char in the string (corresponding to the 17th index of a list representation).

The step cost of any action is 1 regardless of how many spaces a piece is being moved, thus the path cost is simply the number of steps.

Software Design:
The functions Visual, Extract, and Expand are unchanged since the last report.
Visual prints 6 elements of a given argument followed by a newline character until reaching the end.
Extract iterates over a given state and determines the position and direction of each piece on the board.
Expand takes a state and the information of the pieces in it and uses that information to simulate every possible move before returning the moves that result in a valid state and the state itself.
The function Load() has been slightly modified to support given solutions that are on multiple lines, where the previous iteration would only get the first line of the solution.
Compress and Decompress are unchanged and still unused since the last report. Turning all moves into an equivalent number of consecutive 1 space moves and vice versa respectively.

There are several new functions, these being Check, Evaluate and Output
The Check function does exactly what it sounds like; it checks if a given board state is solved or not according to the goal definition provided earlier.
The Evaluate function takes a and returns the number of obstacles between the ‘X’ piece and the right side of the board +1. The +1 is used to represent the fact that the given state is not solved even if there isn’t anything stopping it. This is a vital difference where the A* algorithm is concerned as this function is used as the heuristic.
The Output function appends the contents of a given list to the ‘output.txt’ file.

There are also several instances where sorting a list appropriately needs a custom key.
Asortkey takes a state,moves tuple and returns the the number of moves + the evaluated value of the state. This is used for the A* algorithm to value it’s possible moves.

HillCost uses a slightly more advanced and extreme heuristic than A* as it returns the evaluation of the state multiplied by 50- the number of steps already taken. This means that any step forward is preferred to avoid getting stuck on a plateau, but anything that removes an obstacle is heavily favored.

resultKey sorts results from the search algorithms by length in ascending order, putting the failed results at the back.

The search algorithms implemented are BFS, Iterative Deepening, A*, and Random Restart Hill Climbing.

All algorithms have been optimized to immediately check if a new state is solved as it is discovered to avoid unnecessary calculation.
Similarly all algorithms except for Iterative Deepening include a set of checked states to avoid redundant branches. The Depth Limited Search called by Iterative Deepening has this also, but it does not carry over between iterations.

BFS is implemented almost 1:1 as it is described. It expands each state and keeps searching until it finds a solved state before returning it.

A* is much the same, expanding the first state in the queue, however this time using a priority queue arranged based on it’s heuristic function.

Hill Climbing has to use 2 lists in place of a single set for storing checked states, this is so that it can restart from any of the checked states when it reaches the end of a path.

Iterative Deepening calls a depth limited depth first search until a solution is returned. The depth limited search stops only when it finds a solution or if all states within the specified depth are checked.

(DLS does not appear properly implemented as it not always return optimal solutions)

Algorithm Analysis:
Hillclimb appears to be the fastest algorithm in it's average case, however the resulting solution is rarely optimal or even very good at all. It is also very inconsistent due to the randomness of its operation, and can sometimes take much longer than expected to find a solution.

A* usually appears in a close 2nd place thanks to its informed search, however since the heuristic is very basic, it spends a lot of time exploring a plateau. This is still much faster than a full tree search as seen in BFS and Iterative deepening.

BFS then takes 3rd place as its lack of redundancy leads to a complete search and optimal solutions.

Iterative Deepening is unexpectedly the worst running algorithm by far despite being more advanced than BFS. However this is not due to the algorithm itself, rather it is because of the iterative nature making it search the same states multiple times. Normally this is acceptable because most of the nodes are at the bottom rather than the top, meaning that the redundancy is negligible relative to the size of the problem. This is undone by the fact that all other algorithms specifically avoid redundant trees by using all new states against a set of checked states. This reduces the necessary calculation for something like BFS to only a fraction of its original complexity and allows it to be faster. Iterative Deepening however, requires some measure of redundancy and thus cannot take full advantage of this optimization.

Excluding Iterative Deepening the algorithms as they are implemented can solve any of the given problems within ~6 seconds. Iterative Deepening however additional time and is more easily excluded.

Conclusion:

In summary, although Iterative Deepening is a more advanced algorithm, even a simple optimisation puts it behind even the most basic BFS, and A* Despite being fast and informed is slower than HillClimbing due to it's deliberate nature, whereas Hillclimb is capable of finding **a** solution faster, it is almost never an optimal one. Unexpectedly BFS despite being a simple and naive algorithm can still have it's place for simple uses such as solving a board game like RushHour.