# 2802ICT Intelligent Systems
# Assignment 1 Milestone 1 Report

By Nathan Cowan
S5143344

The purpose of this report is to explain the 'rushhour' style problem proposed by the assignment and detail the structure of the program created to solve it.

The puzzle/board game 'rushhour' is simple in concept and has appeared in many forms. The basis of the game is that there are a number of line blocks of varying length arranged on a grid and they must be moved along their axis in order to clear a path for an objective piece to be moved out of the grid.

There is in this case a 6x6 board with pieces placed on it in the form of 2,3 grid square long blocks in either vertical or horizontal orientation. These are represented by the letters A-K if they are 2 long and O-R if they are 3 long. There is also the special piece represented by X which is always 2 long and must always exist somewhere on the board.

There are 40 example problems and proposed solutions to them in a text file 'rh.txt'. The program created must read this file and extract the relevant information from it. One of a number of search algorithms must be selected and used to find a solution within a given time-limit. Doing this requires that the program be capable of recognising the positions of each vehicle in order to calculate what moves are legal and then repeat this either iteratively or recursively until a solution is found or a specified time limit has elapsed and then convert the solution it might have found into a human-readable form and display it as a series of moves in the form of [vehicle letter][move direction][move distance] e.g. 'DL2'.

The program actually created to fill these requirements has 1 class and 6 functions. Currently there are no implemented search algorithms or the functions for applying a proposed solution and checking if the final state is solved.

The only class is used to represent the positions and movements of each vehicle. There are only 3 variables used: 'id' which is the letter used to represent that vehicle on the board, 'start' which is the position of the top left cell occupied and 'end' which is the position of the bottom right cell occupied. The constructor takes 3 arguments, puzzle, index, and identifier. Puzzle is the current board state, index is the position of the start cell, identifier is as previously stated: the letter used to represent that block on the board. The constructor decides the value of 'end' by looking for the 'id' value in cells to the right and below the 'start' position.

The move() method takes 2 arguments, the board state and the direction of movement. First it checks that the direction is valid for the orientation of that vehicle, then it checks if the space to be moved into is empty. If the space is empty then the vehicle is shifted along in the given direction and the new board state is returned. If the space is taken then false is returned. Returning the new board state or returning false allows the function to be used directly in if statements to both check if a move is valid and to enact it at the same time if it is.

The 6 functions are load(),visual(),extract(),expand() and two functions that are not likely to be used; compress() and decompress(). The functions visual() and extract() are very simple functions which loop over a board state and print a human-readable visualisation of the board or detect where the vehicles are on the board respectively. The function load() opens and reads through the file ‘rh.txt’ and with the use of some simple if statements detects the beginning and end of the example problem states, as well as their solutions and stores each in separate lists. The expand() function takes an initial board state and a list of vehicles and calls their move() function 4 times in each direction using a copy of the argument state. If the result is not false then the new possible state is stored along with the move that it represents. This is the function that will be called by the search algorithms to generate new states. The last two functions are compress() and decompress(), they are designed to combine similar moves into one move and to separate them into the single space unit moves respectively. They are not used in the final implementation as the move function no longer has an argument for how many spaces to move as it did earlier in development.