RushHour Solver

CMPT 225 Project

By Mohammad Bin Zafar Shahid

Computing ID - mba118
SFU ID - 301435375

mba118@sfu.ca

CMPT 225

Simon Fraser University

Summary of The Classes

RadingFile - This class is used to read files as a string to get the boards as a string to give them to BitBoard. If it does not find the file it throws a FileNotFoundException. It is also used to write the output to the solution file.

BitBoards - This class is used to represent a state. It contains three long values horizontalBoard which contains all the cars that can move horizontally, verticalBoard which contains all the cars that can move vertically and a startBoard which contains the value of the starting point of each car. It also contains an object of the class Moves which is used to see which car was moved from where to get the current board from the previous boards.

Moves - This class is used within the BitBoard class to store the moves used to get to the current board. It contains two short values movedFrom and movedTo. MovedFrom holds the index value of the start point of the car from which the last car was moved and MovedTo holds the current index position where the car is positioned. It also has a pointer to its parent Move which points to the previous Move that was used. It is used when we find the winning board to get the list of moves to go to reach the solution.

GenerateBoards - GenerateBoards has a function named allPossibleBoards which produces all the boards that could be made from the current board by using one step. It is repetitively used by BreathFistSeach until it finds the winning board.

BreathFirstSearch - The BreathFirstSearch goes through all the possible boards that can be made by going step by step. It uses a HashSet to avoid any repetition in generating boards. Since it goes step by step using allPossibleBoards from GenerateBoards class it finds the shortest possible solution to a board.

MoveListConverter - This class is used when the final solved board is found. It uses the Move object in the BitBoard of the solved board to get all the set of moves used to go from the starting board to the solved board and converting each of the index movements into the format required in the output file and returns it as an Array List of Strings.

Solver - This class uses all the class above and combine them to get the output solution and write it from the input file. It reads the file and converts it into a BitBoard uses BFS to get the winning Board and passes the Move object of the winning board to The MoveListConverter to get the Steps needed to reach the solution.

Algorithms and Decisions

BitBoards (USED)

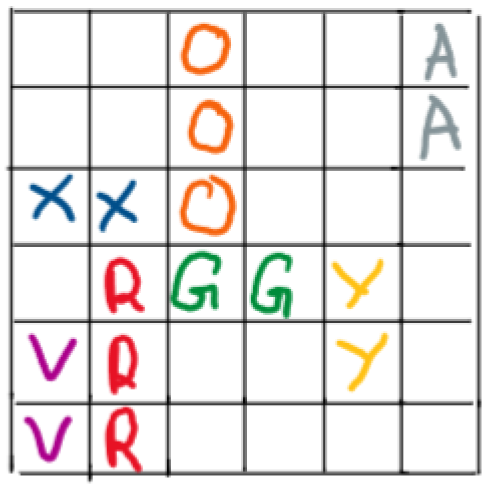
I used Bitboards to store states because it takes less storage space than storing boards as a char array in the HashSet to check for visited states and also makes hashing and checking for equals faster.

The BitBoard class takes a string input and converts it into three long values of 64bit each where the first 36 bits are used to store the index of the cars and there are 28 unused bits.

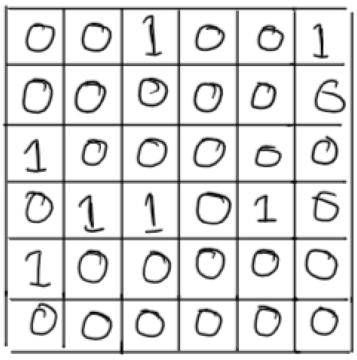
When the file has the board:

```
..O..A
..O..A
XXO...
.RGGY.
VR..Y.
VR....
```

It would look like this on a board :

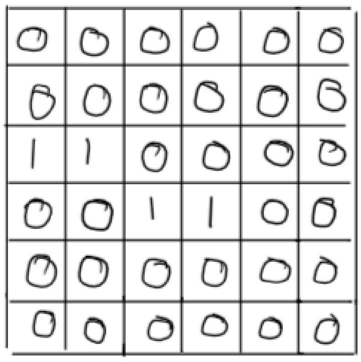


When converted in a BitBoard the first 36 bits of each of the Boards will look like:



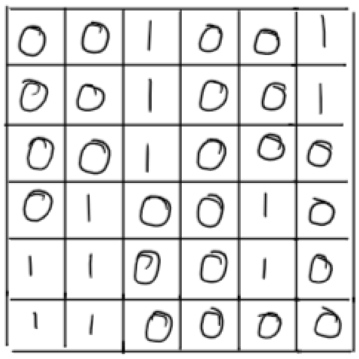
0	0	1	0	0	1
0	0	0	0	0	0
1	0	0	0	0	0
0	1	1	0	1	0
1	0	0	0	0	0
0	0	0	0	0	0

start Board



0	0	0	0	0	0
0	0	0	0	0	0
1	1	0	0	0	0
0	0	1	1	0	0
0	0	0	0	0	0
0	0	0	0	0	0

horizontal Board



0	0	1	0	0	1
0	0	1	0	0	1
0	0	1	0	0	0
0	1	0	0	1	0
1	1	0	0	1	0
1	1	0	0	0	0

vertical Board

The Bitboard stores the data as a one-dimensional array one so in binary they would look like :

```
startBoard:
0010010000001000000110101000000000000000000000000000000000000000
00
horizontalBoard:
0000000000001100000011000000000000000000000000000000000000000000
00
verticalBoard:
0010010010010010000100101100101100000000000000000000000000000000
00
```

The start bit of a car is the index at the top of the car for a vertical car and the index at the very left of a horizontal car. It is mainly used to find the size of a car and makes it easier to move cars around.

All different states have a unique starBoard (because it is impossible to replace the position of a car with a different one without changing to a completely new board) so it makes it easy to store in a HashSet since it is only 8 bytes compared to 36 bytes for storing 36 chars. All startBoards being unique for each state and all the same, states having the same starBoard allows doing equals(obj 0) function by just comparing the starboard (also it should be slightly faster as most CPU use a 64-bit architecture and each long is 64 bits) and also allows producing a hashCode by just using the hash code of the startBoard.

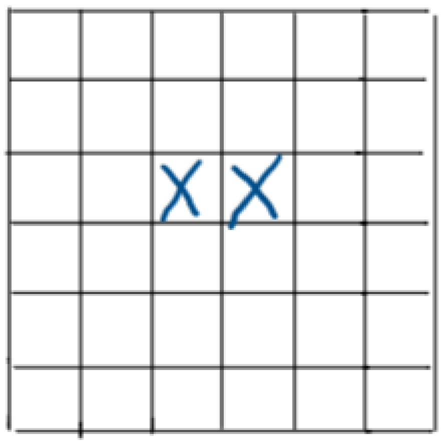
BitBoards also make it very easy to check if a car is horizontal or vertical by checking its presence on the horizontal or vertical Board.

However, it is not necessary or important to use Boards for solving the RushHour Board.

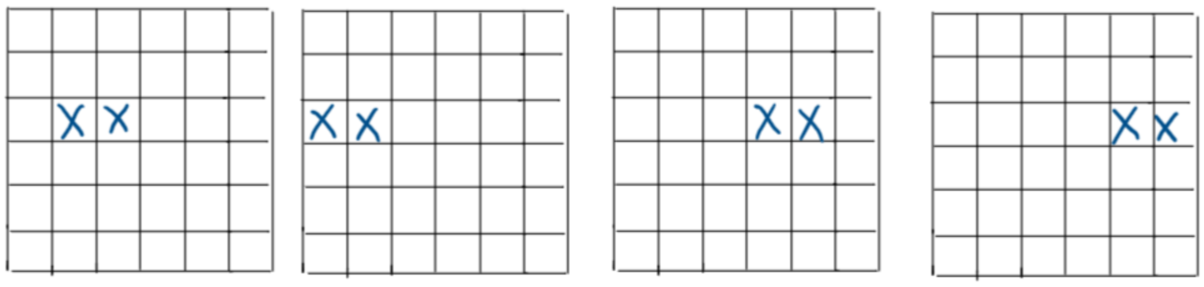
BoardGeneration

allPossibeBoard (Used)

I used all possible Boards to generate new boards which means whenever a board was used as an input it would produce all possible boards possible to be made with just one step except the board that was inputted.



So when this board was entered it would produce :

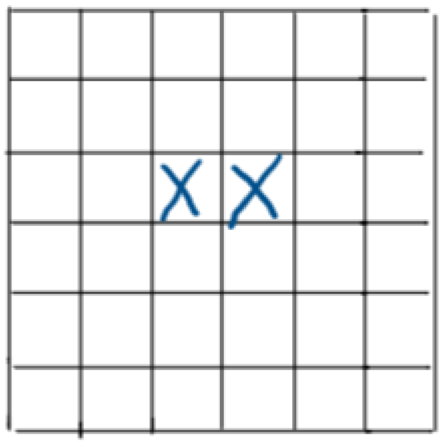


This allows me to go through the entire search Step by Step finding the shortest possible solution.

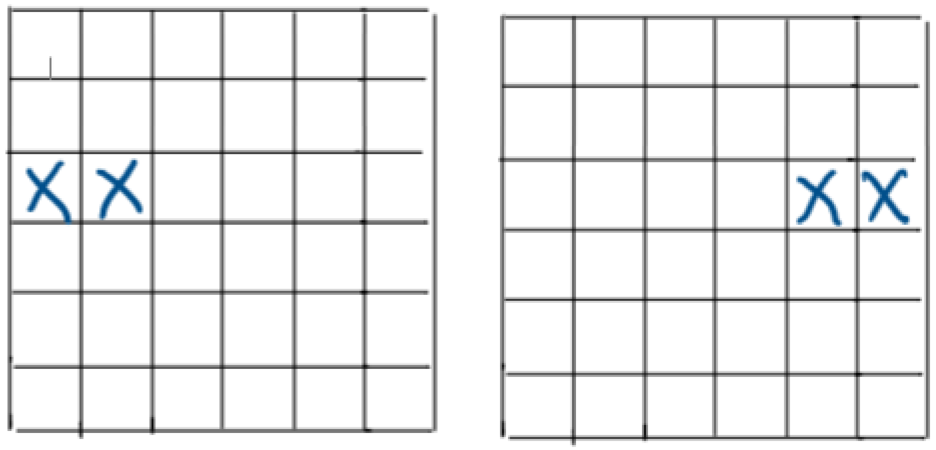
getExtremeBoards (Unused)

Instead of producing all the boards getExtremeBoards just moves to the furthest left or right side and for horizontal cars, the furthest up or downside for vertical cars greatly reducing the number of boards produced and with a few extra steps for some boards and the same number of steps for most boards.

So if the input was -
For getExtremeBoard



when this board was entered it would produce :

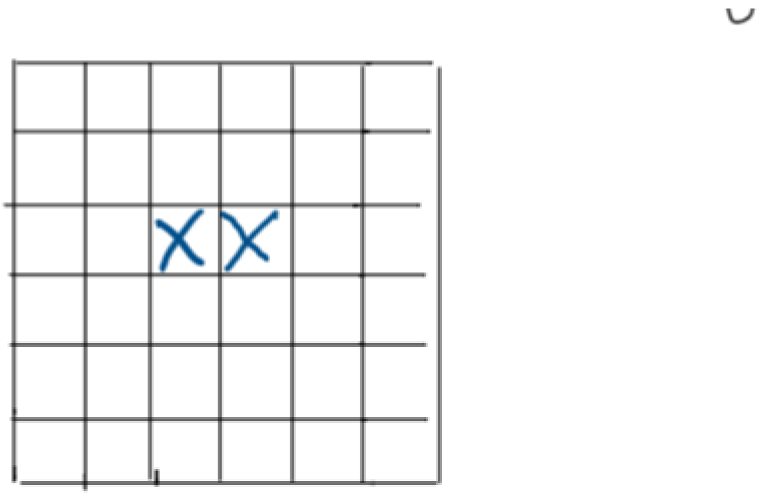


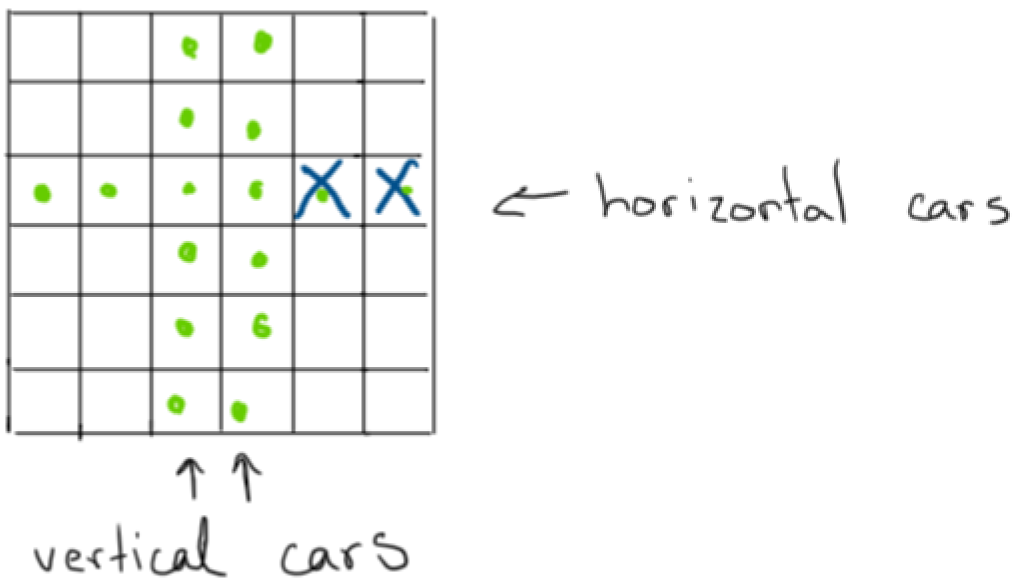
It would do this for all the cars on the board that can move. For all 36 boards, the total number of steps using getExtremeBoards would be 814 compared to 806 when allPossibleBoards is used.

getExtremeBoards helps reduce the number of nodes visited mainly for smaller boards with a lot of free space for cars to move around. I did not use this as the difference was not that big for bigger boards with less free space and there was not much of a time or space issue when using allPossibleBoards.

getSignificatBoards (Unused)

The main reason I implemented this at first is to skip over steps and introduce randomness to my BreathFirst Search. This function would move cars that now can move due to the previous car being moved.





So if the car X was moved to the right in the previous step. The cars which were present in the green dots will have a new position to be in which they could not be previously are moved. So getSignificantBoards only moves the cars that were positioned on the green dots and also depending on if they were vertical or horizontal cars.

I did not use this although it worked incredibly at first is because it skips over a lot of steps. It was combined with getExtremeBoard because it needs a car to already move for it to get cars that have new space to move. Although it usually performs faster it does not give the shortest possible solution and takes more steps than using just getExtremeBoards but can help solve Boards faster.

Nodes (Vertex)(Used)

BitBoards are just used to generate new boards to get moves and to check if we have a winning Board.

The Moves class has been used as Nodes to check how the board got to its current state from the starting board. The moves class works a bit like Nodes in Dijkstra's algorithm as it only points to its parent move. The moves class takes a lot less space as it only stores two shorts one for the index of the previous position(MovedFrom) for the car and one for the current position of that car(MovedTo). They also contain a parent Move which is the previous move used to get the previous board from its parent.

When the solved board is found it takes it Move object of that class and passes it into the MoveListConverter to change the index movement into steps of the output format and return it as an ArrayList.

Search Algorithm

Dept-First Search (Unused)

Initially, Dept-First search algorithm was used to find the solution board. It was faster than normal breath First Search however the number of steps required to reach the solution was very high being in the hundreds for the smaller boards and the thousands for the larger ones. Which was not ideal for finding the solution so it was not used.

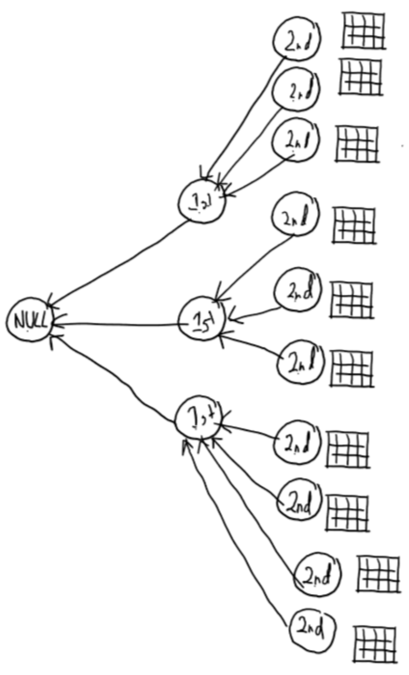
BreathFirstSearch with some randomness and getSignificanBoards (Unused)

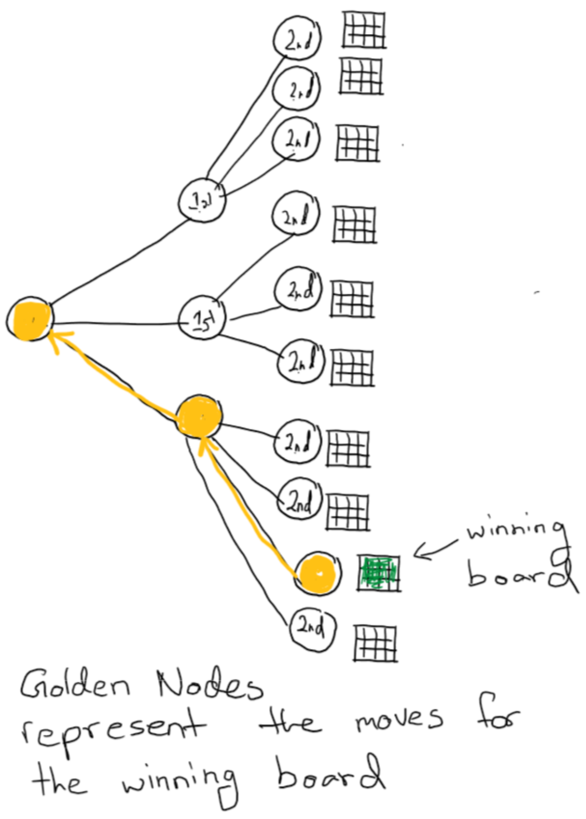
When using randomness with BreathFirstSearch it does not necessarily find the shortest possible solution but it is usually not too far from it unlike Depth-First Search. The getSignificantBoard produces way fewer boards and only produces the most important ones making the search method skip over a lot of steps. It needs to be combined with getExtremeBoards as getSignificantBoards needs a move already used to produce new boards. The more frequently getSignificantBaord was called compared to getExtremeBoards the larger the number of steps would get and the shorter time it would take.

This search algorithm took advantage of the fact that rush hour has multiple solutions at different depths and even when missing one there are other ways to solve it.

BreathFirstSearch Solver (Used)

I used the BFS as it guarantees to find the solution with the shortest amount of steps. I used HashSets to make sure I am not running through a board twice and check if I have already visited a node. HashStes provide a time complexity of O(1) for most time and this helps speed up the time. The size of the HashSet is set 100,000 which is to be extra safe and avoid any HashSet resizing as it takes a lot of time. The search algorithm checks if any node is visited before generating any board from it removing all repetition and also checks whether any board is visited before putting it in the queue. The HashSet only stores the startBoard of the BitBoard as every board has a unique HashSet and it is used to check for visited States(BitBoards). It uses a queue for the Breadth-First Search as it is first in first out and uses a LinkedList for adding boards to the queue.





The Checker Box represents the BitBopards which are stored in a queue and the circles are the nodes (The moves used to get to the current BitBoard). When a winning BitBoard is found. It Moves and its parent Moves are used along with MoveListConverter to get the steps for the final solution.

Solver (Used)

The Solver uses all of the classes to get the solution board. It reads the String using the ReadingFile class and turns it into a BitBoard. Then it passes the BitBoard into the BreathFirstSearch.BFS function to get the solved board. It takes the solved Board and takes it moves class to pass it onto the MoveListConverter. Where the MoveListConverter returns an array formatted into the string which is written on the output file,

Steps and running time

The algorithm solves all 36 Boards provided in the project with a total of 806 steps in 310ms in CSIL (I did not write the solution down in a file but did generate all the solutions step-by-step).