CMPT 225 FINAL PROJECT

Rush Hour Solver

Shintaro Morikawa 301435113
Chris Wen 301251957

Explanation of the classes used:
For our solution, we used two classes:

- A RushHour class that implements the Rush Hour board as a String and applies the BFS algorithm to find a solution.
- A Solver class that prints the solution to a .txt file (i.e. A00Sol.txt).

Inside the rushhour class, we have a fully functional Rush Hour game that generates new board states until a solution has been found (X car has moved all the way to the right of the board). We then append the moves that were used to get to the solution to a private member ArrayList and print each element of the ArrayList to the solution file.

The Solver class calls functions inside the rushhour class to solve a board from a specified file and then print that solution to a file name specified by the user. We treat each move of 1 square as a separate move so the solution file will print out something like:

```
AD1
AD1
```

instead of:

```
AD2
```

Data Structures Used:
The data structures used in our solution include:

- One HashMap to link the different board states between car moves.
    - This is constantly being updated during BFS and is needed to keep track of which moves lead to which board state.

- A Queue for the BFS algorithm to keep track of the board states that have already been visited.
    - This is important to not revisit board states we have already seen. This results in an improvement in the algorithm’s runtime and a reduction in the total number of board states in order to save on time and memory.

- One ArrayList to store the moves that lead to a solution.
    - This is used by the Solver class to print out the moves necessary to reach the solution board’s state front the original unsolved board state.

Algorithms Used:

For our final working solution, we used the BFS algorithm. Our algorithm manages to solve all of the puzzles given in the assignment files almost instantly. For the BFS algorithm, we used a queue to keep track of the board states that have already been visited and incrementally generate the graph as we move cars around until a solution has been found. We then print out the route ArrayList inside the rushhour class that contains the path needed to go from the original unsolved board state to the solution board state found by the BFS algorithm.

Difficulties in Creating the Solver:

In our initial implementation, our A* algorithm was creating far too many nodes due to a bug inside our generateGraph() function which ended up slowing down our program. The most amount of time was spent debugging a very complicated function (200+ lines of code) before failing to find it and starting from scratch with a BFS implementation.

When implementing the BFS algorithm, the most time consuming part was figuring out how to print the path to the solution board state once we have found a solution. We found a way to do this recursively using the numMoves() function, which traces the car movements to the beginning board state once a solution has been found.

Optimizations:
In our initial implementation of the BFS algorithm, we used an ArrayList to store the board, where each element of the ArrayList is a row of the board, represented as a String. After implementing some member functions, we realized that this is pretty inefficient as we found ourselves having to traverse through every element of the ArrayList and also every character inside each element’s charArray. This is very slow and very noticeable when creating thousands of board states. We then switched to a full String representation and used parsing techniques implemented in the rowTrans() function to convert it to a workable board format, which was much more efficient and noticeably reduced our runtime.

Omitted Code:
Initially, we implemented the A* algorithm with a simple heuristic that will be discussed in the next section. After many bugs with the algorithm and poor runtime even for the easiest of puzzles, we switched over to the BFS algorithm and for the most part, started from scratch as our heuristic or implementation of the A* algorithm did not seem to be working very well. After switching to the BFS algorithm, our runtime improved significantly and were solving all puzzles almost instantly which likely means that the implementation provided in the A* algorithm had a logical error or may have overestimated the number of moves needed for some board states to reach a solved state, thus resulting in nonfunctional code.

A* Heuristic Analysis:
When we implemented the A* algorithm, the following heuristic was used:

- We had a cost() function that underestimates the number of moves needed for the board to reach a solved state.
    - Inside this cost function, we considered the freedom of movement of each car (0 freedom of movement if it is unable to move due to one or two cars blocking it, 1 if it can move in 1 direction but not the other, etc).
    - We then summed the freedom of movement of each car on the board and compared this value to the total number of cars on the board.
        - If this value was less than the total number of cars on the board, we concluded that the board was pretty complex and each car would need to move at least once in order to reach a solved state, so we set the cost equal to the total number of cars on the board.
        - If this value was more than the total number of cars on the board, we concluded that the board was of easy/medium difficulty and at the very least, a third of all the cars may be needed to move at least once to reach a solved state, so we set cost equal to a third of the total number of cars, rounded down.

Task Delegation:
(Only work for the submitted code is included. We spent more time on the A* algorithm that was too slow)
Individual work:
Chris:
rushhour constructor
makemoves()
design doc
file IO

Shintaro:
boardDiff()
generateNewNodes()

Pair Programming:
numMoves()
BFS algorithm
debugging
Solver class integration
most of the helper functions used inside the rushhour class