Ross Forster 301319236
Omar Elshehawi 301388741

Project Documentation

Our Thought Process whilst completing the project:

In order to make our BFS work, we need a getNeighbors function. The getNeighbors allows us to get every possible move on the board. In order to make our getNeighbors to work, we need a makeMove which moves a car in a direction with a specified length. We had to create a checkLegalmove function to make sure the moves we are making are possible whilst our getNeighbors attempts every possible move. All these functions needed the carNames and their directions so we created a helper function, setup(), that inserts all the names of the cars and their respective directions into lists for later access. After every possible move is recorded as a board and inserted into a list, we created our solve() function that uses BFS to find our answer. We set the parent board as a node to be able to access previous nodes throughout our tree. Also implemented a Hashmap to make sure no boards are visited more than once. Our solve function attempts every possible board until it finds a solution.

Q: Explain your choices of classes, algorithms, heuristics etc.
A: Our project consists of 2 classes, the Solver class which implements our Node class. Our Node class has within it a board, prevNode which contains the board from the parent node, and prevMove which contains the move to go from parent node to child node. Furthermore, our Node class has getters and setters to allow access within our solver class.

Q: Was one of the heuristics always better? Or different heuristics solved different puzzles?
A: We only used Breadth-First Search so we didn't use any of the provided heuristics.

Q: Describe which parts were easier and which required more time.
A: Our getNeighbors function took the most time as it required multiple helper functions in order to work. We had to make the makemove function, our setup, and our checkLegalMove function just for getNeighbors. The easiest part was the solve function as it was a clear implementation of the BFS algorithm which was fairly simple to understand.

Q: Describe what you wrote, but didn't include in the final project.
A: We wrote a print function as well as made a list for the position of the cars on the board. We did use print but left it out of the final project because we don't need it. We made a list of the car positions but we didn't use it so we took it out.

Q: How did you divide your work?
A: Both of us worked on every section of the project together.

Ross Forster 301319236
Omar Elshehawi 301388741

Representation of our pseudocode and rough work:

Rushour Algorithm:

Import Board ---Complete

queue - all initial boards(nodes)

output():
-have each node store the previous node and the previous move
-takes final board,compares it to prev board, spits out the move

General Idea:
solveBoard();
-algorithm (BFS)
-
isSolved(board);
-Returns true if XX is in [5][2]
Node: comparable interface needed
-board

List<Node>getneighbours()
-check one move ahead for everypiece
-each move constitutes a new node that is added to the queue
Boolean checkLegalMove(carName,length);
-checks if car can move based on direction and length
-checks if any cars are in the way
makeMove();
-Specify all states of cars to find the correct state that solves board
-BFS
-
Data:
Car name
Car length
Car dir
Car pos

SetUp
-nested for loop
-add car names(no duplicates)

Ross Forster 301319236
Omar Elshehawi 301388741

-check direction
-count length of car
-continue

(position) Cars: AA(1) BB(0) CC(0) DDD(0) FFF(0) GG(0)
11112 pos++
01110 pos(i,j)
00100
00010
00000
00000

Hashmap:

```
HashMap<Integer, Integer> Positions = new HashMap<Integer, Integer>();
```

Getneighbors moves every car 1 step each direction and outputs board for every single move

20 board states
AA 1 step left-board state
getneighbors AA 2 step left-board state
...….
..A….
...A...
……..
……..

A [0][1]---[0][2]
A[0][1]----[0][3]