Rush Hour Solver

CMPT 225

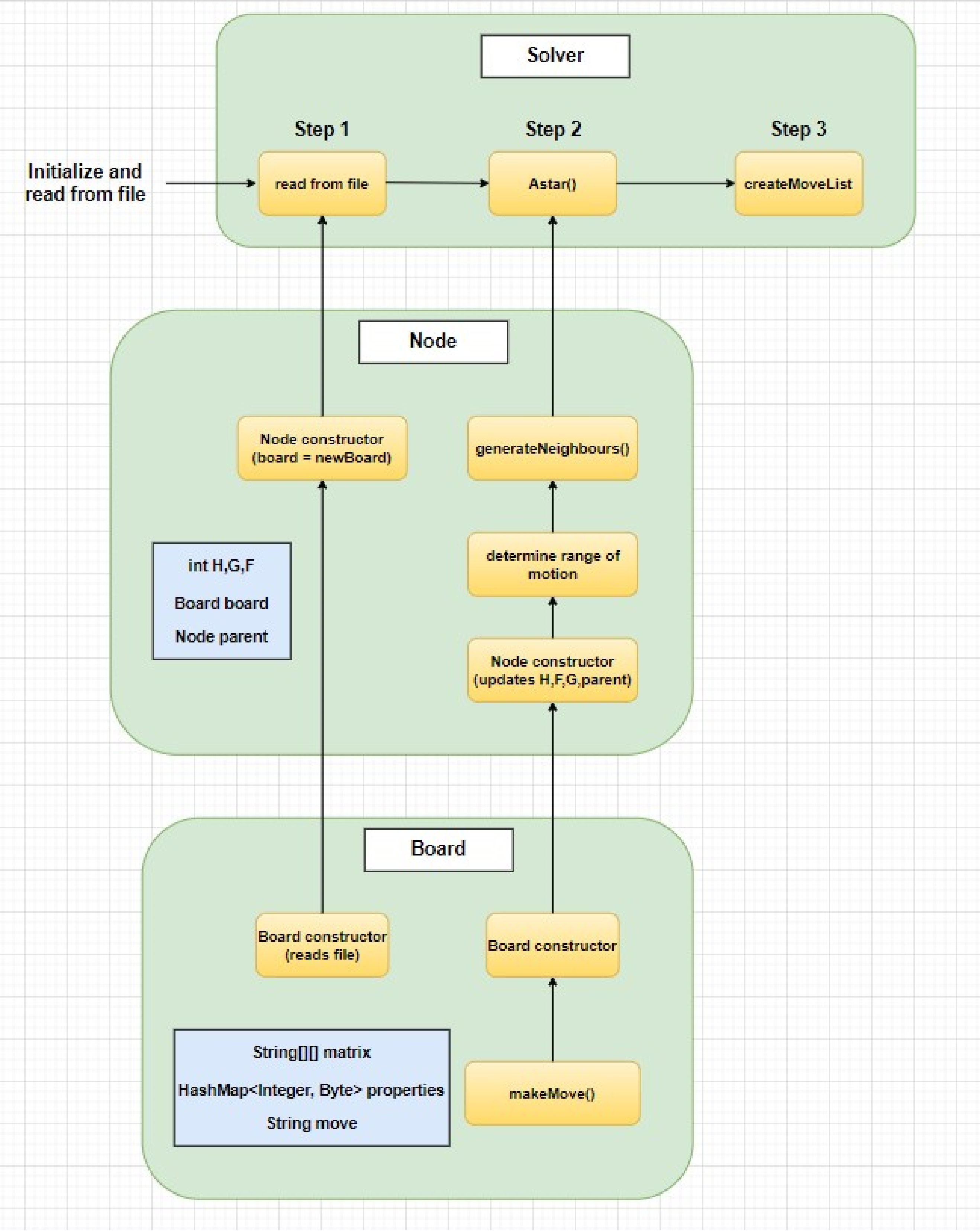
IGOR SHINKAR

Rafael Guevara- 301410015

Golrokh Nouri - 3013711110

1. **General outline/class structure:**

Below is a high level code structure showing the flow of code and the responsibilities of each class (note: not all methods and members are shown here.


Solver
Step 1
Step 2
Step 3
Initialize and read from file
read from file
Astar()
createMoveList
Node
Node constructor (board = newBoard)
generateNeighbours()
determine range of motion
int H,G,F
Board board
Node parent
Node constructor (updates H,F,G,parent)
Board
Board constructor (reads file)
Board constructor
String[][] matrix
HashMap<Integer, Byte> properties
String move
makeMove()

Step 1:
Solver reads from the file. This happens by calling the Node constructor which initializes all of its members with default values (null/zero) and initializes its board member through a Board constructor. This constructor reads from the file and initializes the matrix and fills out the properties of each car (orientation, length).

Step 2:
Solver now runs the Astar algorithm. This includes the OpenQueue and ClosedSet that traverse the graph using a BFS/light heuristic implementation. The function calls the generateNeighbours method of the Node class (which importantly, returns but does not store the neighbours) of the current node. From there, it must determine the range of motion of all the cars in the board (how many moves each car can make). Then, it goes through all the moves of all the cars, calling the Node constructor to make a new node at each new move. This constructor updates H,F,G, and the parent and calls the Board constructor to make a copy of the old Board with one move executed (stored in the Node's member: board). The move is stored within this board, illustrating how to get from its parent to itself.

Step 3:
Once the algorithm has found a target node, the createMoveList() method traverses from the solution to the beginning through the nodes' parents, concatenating the list of moves in reverse order using a stack

2. **What we left behind:**

- When writing the board class initially, we thought to optimize space by making the Board class a helper class with all static variables, but then decided against it since its elements needed to be accessed and changed.

- We thought about going through the open queue everytime we generate a node's neighbours to check for uniqueness. The goal was to pay the price of O(|OpenQueue|) in exchange for uniqueness of nodes. However, when going through the algorithm again, it was unnecessary, and we could get the same results from having multiple nodes representing the same board in the open queue. If we pull out a node from the open queue that has been processed already (therefore is moved to the closed set), we will discard it.

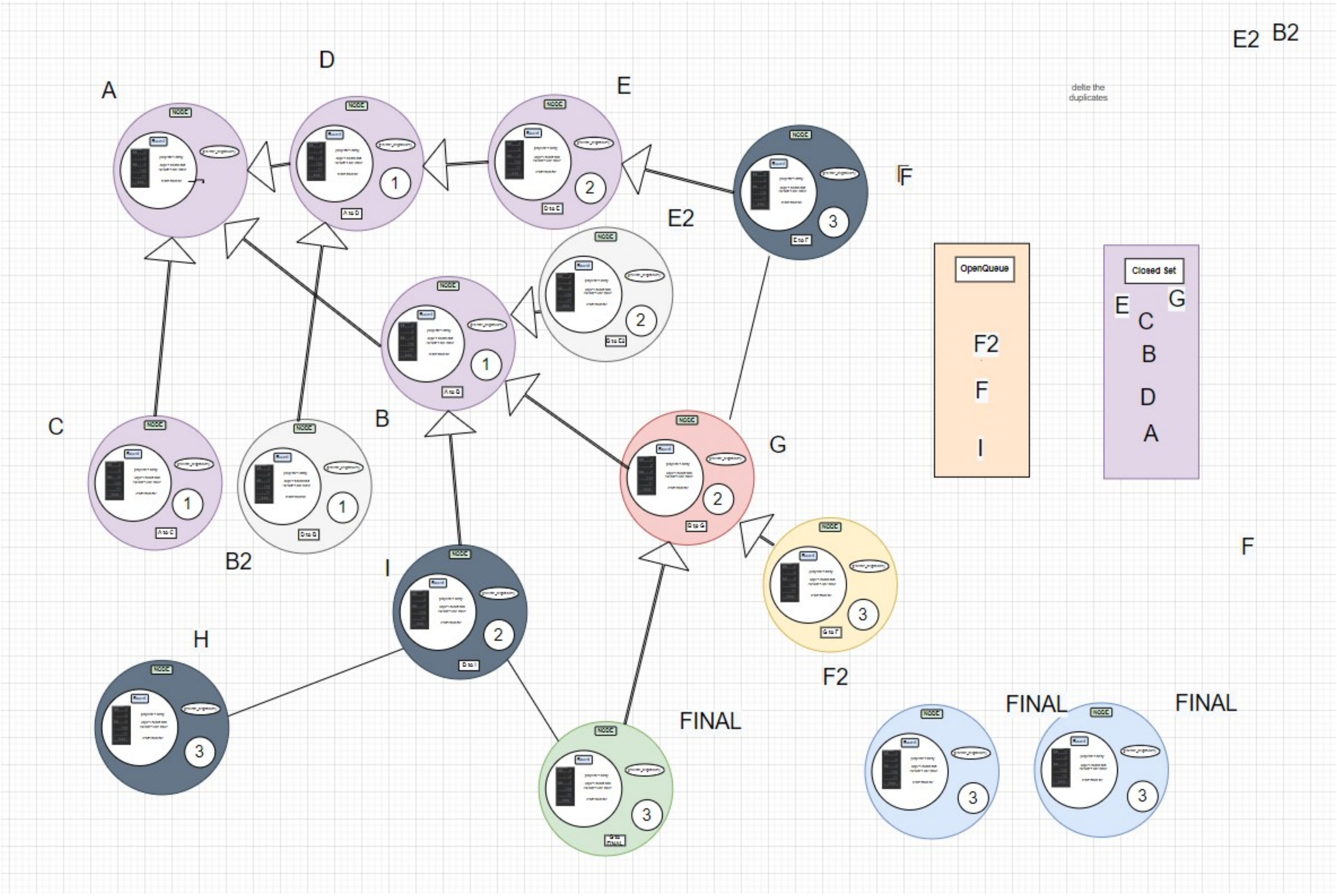
- We thought of implementing a graph class representing all nodes and an edge class representing the path that led the nodes there (e.g. X car moved right 1 step to get to this node). This was to solve the issue with uniqueness and help track our steps back to the initial node once a solution has been found. We then realized given the previous point about uniqueness, each node would only be given one parent (even if they mathematically have multiple). This means we could store the move describing how to get from ‘parent’ to ‘child’ in the child node.

3. Difficulty of tasks:

- Coming up with a structure for the classes and eliminating unnecessary classes was the most challenging part.

- Integration. We wrote the Node and Board classes and then went to integrating/implementing them in the Solver class. Some unit tests were written, but problems did occur and debugging became harder during integration

- Visualizing the algorithm in use with the rush hour board was difficult at first. This led to most of the problems discussed in section 2 of this document. Once we created diagram ‘models’ of our algorithm and worked to see what would happen there, we could generalize the idea and implement it in code. A diagram is shown below to illustrate our thought process (details are not meant to be accurate in this picture as this was a WIP diagram used to help us figure out the algorithm)



4. Choice of Data Structures and Algorithms (the main ones):

- HashMap for ClosedSet. A Hash Map was used to minimize the contains function. We had to override the equals function by hashing the arrays that make up the board in our Board class.

- Priority Queue for OpenQueue. Used to get the minimum value (based on f) in constant time. We had to override the compareTo function and have Node implement Comparable by comparing two Nodes based on their F's.

- Stack for creating the movelist. Once we found a target node, we would traverse back to the beginning through the node's parent. A stack was used so that the moves would be reversed to normal order.

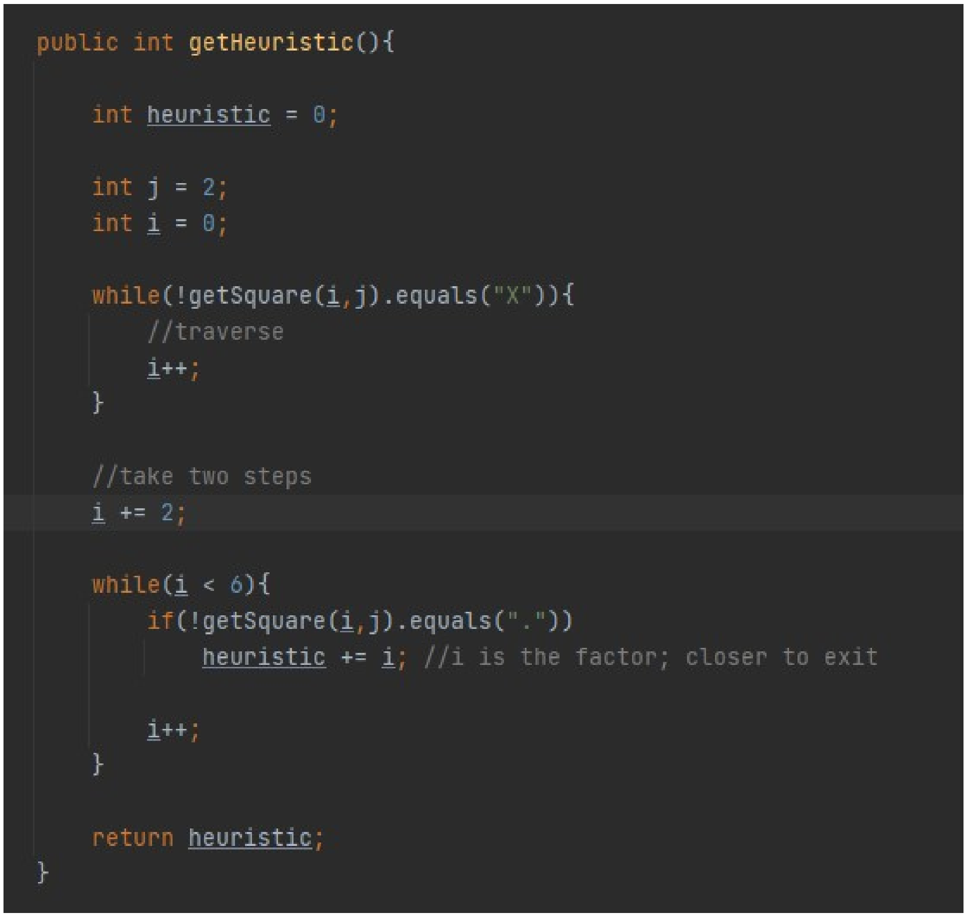
5. Software Optimization:

- Adding a getLength function for each car helped save a lot of time when it comes to calculating the range of motion for each move.

- Never storing the neighbours of a Node. generateNeigbours() simply computes and returns all neighbours of a Node

  After completing the program and running it with the board sets A,B,C,D, and an additional 5 more difficult boards courtesy of a student on piazza, we got a runtime of about 4 seconds (through pure observation). Then, we tried to optimize where possible to see how much further we could bring down the runtime. With these changes, running the same boards gave us a runtime of about 2 seconds. Based on some light testing and analysis, each change is listed on the next page in order of effectiveness.

- Heuristic implementation. The number of non empty spaces in front of the car would be proportional to the heuristic. The closer the square to the heuristic the more weight it carries. This is simply the x coordinate of that square.

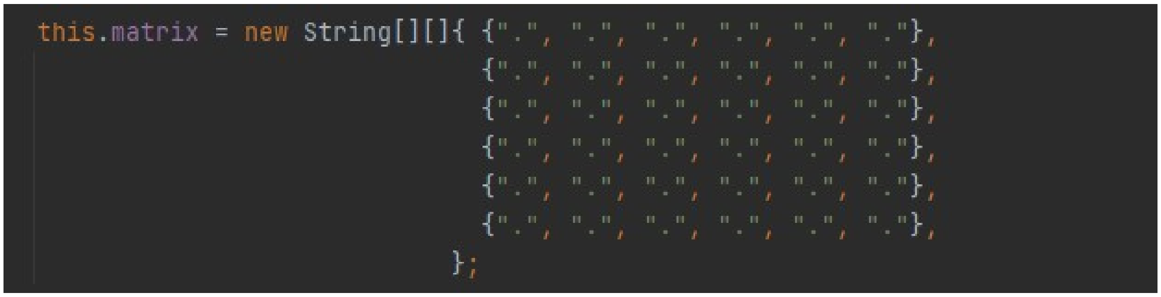


```
public int getHeuristic(){

    int heuristic = 0;

    int j = 2;
    int i = 0;

    while(!getSquare(i,j).equals("X")){
        //traverse
        i++;
    }

    //take two steps
    i += 2;

    while(i < 6){
        if(!getSquare(i,j).equals("."))
            heuristic += i; //i is the factor; closer to exit

        i++;
    }

    return heuristic;
}
```

- Make visitedCars local to makeMove and change to a HashMap. There was originally a ‘ClosedSet’ type member in the Node class that told us which cars we had visited when generating neighbours. This was implemented as an ArrayList of Strings. This variable is now made local to the generate neighbours method and turned into a HashMap<Integer,String>. This would allow for constant time “contains” calls and affects the runtime of generating neighbours (which is called for every node!)

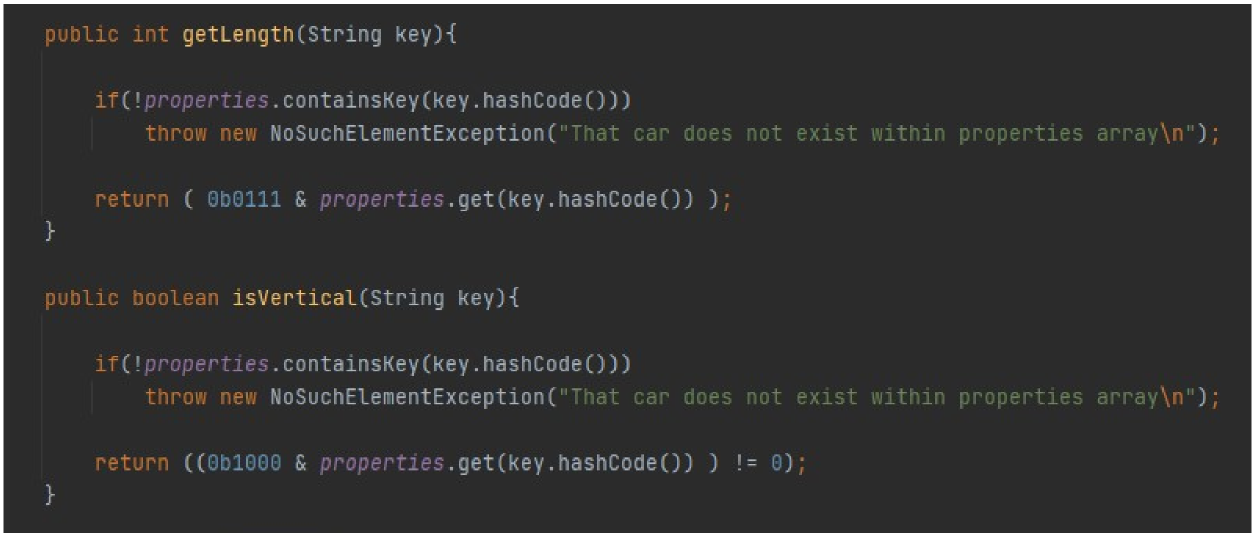
- Store hashCode during construction so the hash function is called only once. Saves computation when comparing nodes in the ClosedSet

- Initialize the matrix beforehand, only modifying it if we find a car. Originally, we would read from the file and read every single character to add to the matrix. Now, reading from the file consists of only processing data when a car is spotted. This would reduce runtime during initialization for boards with lots of empty space.



```
this.matrix = new String[][]{ {".", ".", ".", ".", ".", "."},
                              {".", ".", ".", ".", ".", "."},
                              {".", ".", ".", ".", ".", "."},
                              {".", ".", ".", ".", ".", "."},
                              {".", ".", ".", ".", ".", "."},
                              {".", ".", ".", ".", ".", "."},
                            };
```

- Properties 'array' using Bytes and the use of bitmasking. The original implementation used a HashMap<String,int[]> called properties to store the orientation and length of each car. This would require 1 byte for the key (the string will only have one char) and up to 4 bytes for the value. Instead, we used a HashMap<Integer,Byte>. Then we would hash the car name and use that as the key and store both orientation and length in a single byte. Bit masking is used to both get and set the values (bitwise & and bitwise | respectively). The first three bits represent the length and the fourth bit tells us if it's vertical (1 means yes). For example, 0b 1 010 means (vertical, length:2). This saves both space and potentially some practical runtime as computers in general are great at working with bytes.



```
public int getLength(String key){

    if(!properties.containsKey(key.hashCode()))
        throw new NoSuchElementException("That car does not exist within properties array\n");

    return ( 0b0111 & properties.get(key.hashCode()) );
}

public boolean isVertical(String key){

    if(!properties.containsKey(key.hashCode()))
        throw new NoSuchElementException("That car does not exist within properties array\n");

    return ((0b1000 & properties.get(key.hashCode()) ) != 0);
}
```

6. **Work split:**

- The first step was to meet and come up with a general outline of what goes in what class. We did this together and decided that we needed a Board class, Node Class, and a Solver class.

- We then each worked on one class. Golrokh wrote methods for the Board class and Rafael for the Node class.

- The subsequent few meetings consisted of testing small methods and ensuring their correct output.

- After that, we decided we needed a graph and edge class and a few days were spent working on implementing that. However we eventually decided they were not necessary so they are not included in the final implementation of the code.

- The Solver class is implemented mostly by Rafael with Golrokh implementing a couple of the methods.

- Most unit/integration tests are written by Rafael.

- Golrokh wrote efficient algorithms for methods like range of motion of each car, implementing a map of properties for each car (car name → length, orientation), makeMove.

- Rafael made small optimizations to the code to increase runtime efficiency and wrote a basic heuristic implementation.