COMPSYS202/MECHENG270
Assignment 4 - Traffic Simulator

Partha Roop and Jin Woo Ro

Due: October 23 11:59 pm, Weight: 20%

1 Introduction

Traffic simulation plays a key part in designing modern transport infrastructure. Microscopic simulators simulate the physics of individual vehicles, while macroscopic simulators abstract the detailed physics by some abstraction. This helps in creating scalable models, even for simulating a whole city's traffic. One such macroscopic model is known as **Traffic Cellular Automata**. Here a segment of a road is divided into a set of lanes and each lane is further divided into a set of cells. This is a *space discrete* and *time discrete* model i.e. the space is divided into discrete segments called *cells* and the simulation time is broken down into discrete steps called *ticks*. During every tick, a car can move forward one step to follow the car ahead. This is known as *car-following*. Alternatively, it might want to change lane to an adjacent lane, which is known as *lane-changing*. Either a car following forward movement or lane changing perpendicular movement is allowed as long as there are no conflicts i.e. the cell to which the car intends to move is free. Figure 1 provides an example of the cellular structure of a set of lanes. Figure 4 illustrates the two types of movements by an individual vehicle.

In this assignment, you will program a simple traffic simulator, which implements the car-following and the lane-changing dynamics. You are required to use linked-lists as an underlying data-structure to store the vehicles being simulated. You are free to choose either a singly or a doubly linked-list. The operation of the traffic simulator is as follows. We start by defining some terminology, which will be used throughout this document.

- Lane: We consider a set of cars to be travelling in lanes, which are indexed from $0 \leq k < n$ to simulate a road consisting of n lanes of a highway. We will call $k = 0$ as the first lane.
- Cell: Each lane is divided into equally spaced cells, where each cell can occupy at most one car.
- Platoon: A group of cars that are occupying an individual lane are known as a platoon.
- Tick: The simulation time is discretised into a set of ticks. During every tick, each vehicle attempts to perform either a car-following action or lane-changing action. These

actions are enabled in case of no conflicts i.e. when the cell to which they want to move is free. When both these actions are enabled, the lane changing action takes priority. Once every car has performed their movement actions, the simulation grid is updated. This behavior is repeated until the end of the specified number of ticks is reached.

First, the user specifies an input text file, which contains the initial state information of all cars. Second, the simulator computes the position of each car during the current tick. Third, the positions of cars in each tick are stored in an output file. This process continues until the specified number of ticks needed for a given simulation in the input file. Finally, the output file is then animated using the provided visualiser.

1.1 Learning Outcomes

From this assignment, students will learn:

- You will be able to understand different types of data structures such as arrays, vectors and their distinction with linked lists, for this assignment.
- You will implement a linked list based system for simulation of traffic cellular automata-based traffic simulator.
- You will model the vehicle dynamics based on linked list operations.
- You will develop a good understanding of software design using the model-view controller (MVC) pattern.
- You will also develop good understanding of the traffic simulation using simplified cellular automata.

1.2 Getting Started

To start this assignment, follow the instruction below:

1. Download the skeleton code from Canvas, and unzip it in a folder.
2. Open a terminal, and execute `make` for the compilation.
3. Run the program through `./simulator.o` on the terminal. The program automatically reads a specific input file called *input.txt* file, and generates *result.txt* file. Check if the output file is generated.
4. We provide a Qt project, which can animate the output file. For this, follow the guide in Section 3.

The next section explains the design of this assignment.

2 Traffic System Design

We consider a grid map, where the positions of cars are captured in discrete spaces as shown in Figure 1. The cars obey the following basic rules:

1. Cars always move from West to East (i.e., left to right).
2. The top lane is the first lane, and the bottom lane is the last lane. Considering the direction of moving cars, lane-changing to the right (in the real-world) is equivalent to moving to the lane below in our simulation.
3. Each car can move at most one cell for each unit time interval *tick*. Essentially, car-following is moving forward by one cell, and lane-changing is moving to the side cell. However, in our assignment, only one action can be taken at a time, because the cars cannot move more than one cell at a time.

2.1 Car State

The state of a car can be expressed using a four-tuple as follows:

$$VehicleState = (ID, Lane, Pos, TurnSignal) \tag{1}$$

ID is a unique positive integer for each car. *Lane* is also a positive number representing the lane index, where 0 indicates the first lane. *Pos* is the car's horizontal position. Since the cars are moving from left to right, the position value increases by one as they move to right every tick. Lastly, $TurnSignal \in \{0, 1, 2\}$, where 0 means that no turn signal is active, 1 means that the left turn signal is active, and 2 means that the right turn signal is active. For example, consider the car with id 1 in Figure 1. The state of this car at tick 1 can be written as: $(1, 0, 3, 0)$ i.e. car 1 in lane 0 has its turn signal off.

2.2 Input/Output File Format

The program specifically reads the file "input.txt". An example is provided in Figure 2. The first line indicates the maximum simulation tick. For example, 15 means that the simulation terminates after executing tick 15. From the second line onwards until the "!"-character is detected, the initial position of each car is specified. The second line represents the first lane, the third line is the second lane, and so on. In each line, the numbers are in the format $id1, pos1; id2, pos2; id3, pos3; ...$, where the first number before the comma is the ID of a car, and the second number is the position. For example, in Figure 2, consider the second line: "1,2;2,4;3,8;". From this line, we know that, in the first lane, there are 3 cars with ID 1, 2, and 3 with the initial position of 2, 4, and 8, respectively.

After the exclamation mark in Figure 2, we have 3 numbers separated by commas in each line. Each line is called a *lane change command*. The first number indicates the tick count at which this lane change command needs to be executed. The second number is the car ID. The last number is the turn signal value (0 = no light, 1 = left light, 2 = right light). For example, a lane change command $0, 1, 1$ means that, in tick 0, the car with ID 1 wants to change lane to the left.

The output file "result.txt" has a specific format as shown in Figure 3. In each line, the first number is the tick count. Then, within each set of parentheses are numbers representing the state of a car as described in *VehicleState* (see Equation 1). Note that from top to bottom, the state of *all* cars are given in chronological order.

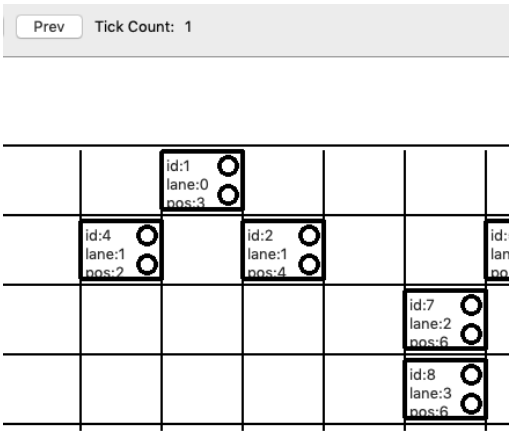

Figure 1: Simulation snapshot.

```
15
1,2;2,4;3,8;
4,1;5,6;6,8;
7,5;
8,5;9,7;
!
0,1,1
0,2,2
2,7,2
4,6,2
5,9,1
8,5,1
```

Figure 2: Input file format.

```
0;(1,0,2,1);(2,0,4,2);(3,0,8,0);(4,1,1,0);(5,1,6,0);(6,1,8,0);
1;(1,0,3,0);(3,0,9,0);(4,1,2,0);(2,1,4,0);(5,1,7,0);(6,1,9,0);
2;(1,0,4,0);(3,0,10,0);(4,1,3,0);(2,1,5,0);(5,1,8,0);(6,1,10,0);
3;(1,0,5,0);(3,0,11,0);(4,1,4,0);(2,1,6,0);(5,1,9,0);(6,1,11,0);
4;(1,0,6,0);(3,0,12,0);(4,1,5,0);(2,1,7,0);(5,1,10,0);(6,1,12,2);
```

Figure 3: Output file format.

2.3 Single Car Movement Rule

There are one of two possible actions that a car can take during any tick: (1) car-following and (2) lane-changing. The car-following action means that a car will move forward by one cell. This action can be triggered only if the front cell is empty. On the other hand, the lane-changing action means a car will attempt to move to a side cell. This action can be triggered if there is a lane change command and the adjacent cell is empty in the suitable lane to which the car wants to move.

Figure 4 shows the car-following action and the lane-changing actions. Since both the actions cannot be triggered simultaneously, we must prioritise the lane-changing action. This means that, if a car can perform both actions, then only lane-changing is triggered. If lane-changing is not possible, the car will perform car-following and move ahead one cell. A few important notes are:

- If there is a lane change command, which makes a car to move outside the map, then this command should be ignored. For example, a car on the first (top) lane cannot left lane-change, since this is the left-most lane.
- We do not allow multiple lane change commands for a car in a single tick. For example, we do not allow 0, 1, 1 (tick 0, id 1, left lane change) and 0, 1, 2 (tick 0, id 1, right lane change) to be specified together in the input file.
- We assume that only the correct lane change commands are specified in the input file.

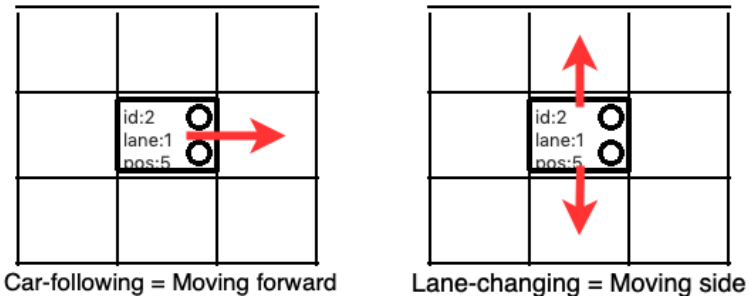

Figure 4: Car actions: car-following and lane-changing.

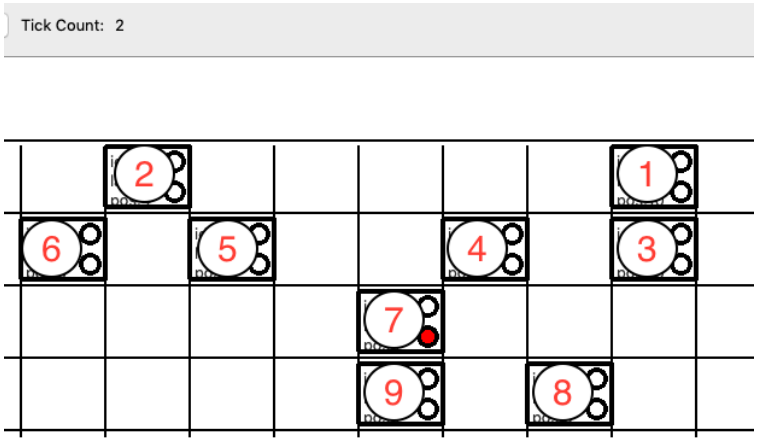

Figure 5: Execution order of multiple cars.

2.4 Multiple Cars Movement Rule

The cars are updated per tick through a sequential execution based on a specific order. More specifically, the cars in the upper lane are executed first, then the cars in the lower lane are executed. Within a lane, the right-most car is considered first, and then the next car to the left is considered. Figure 5 shows the an example case and the order that cars are considered in, with the order represented by numbering within the circles.

Such a sequential execution naturally resolves potential collisions between cars. For example, consider the case in Figure 6a. At tick 1, there are two cars attempting to lane change to the same lane at the same time. If the actions of these two cars are executed simultaneously, this situation should result in a collision. However, if the cars are executed sequentially, the car with ID 1 is executed first as it is located in the upper lane. Since the cell next to it is empty, the car with ID 1 can successfully perform a lane change. Then, the car with ID 2 is executed. By this time, the cell it wants to move to is already occupied by the first car. Therefore, car 2 triggers the car-following action instead of the lane-changing action. As a result, the simulation results in no collision at tick 2 as shown in Figure 6b. In this assignment, the traffic simulation strictly obeys the sequential execution rule.

(a) Cars at tick 1.

(b) Cars at tick 2.

Figure 6: Collision is naturally avoided in the sequential execution.

3 Qt Visualizer Setup

This section explains how to compile the traffic visualizer from the source code provided on Canvas. Note that, Qt visualizer is only an option for testing your program. If you don't want to use it, you may validate your program by directly checking the output result.txt file. To use the Qt visualizer, you need to download the visualizer source files on Canvas, and install the Qt framework from its official website. Follow the instruction below:

1. Go to `https://www.qt.io/download`, and download Qt for your machine (window, mac, and linux).
2. While installing Qt, make sure that Qt is actually included in the installation. For example, during the installation on Mac, you need to tick "macOS" checkbox as shown in Figure 7. If you do not do this, Qt will not be installed correctly. Somehow, this checkbox is not ticked automatically (at least on Mac), so make sure that you have this box checked and are correctly installing Qt.
3. After the installation, open the software called Qt Creator, located in the folder where Qt is installed.
4. Meanwhile, download the traffic visualizer source code from Canvas, and unzip it in a folder.
5. Open Qt Creator, and go to the file menu at the top. Click "open file or project", and select the project file (.pro file) that you downloaded.
6. Then, Qt Creator might ask you to choose a kit, which is basically a compiler for Qt. Select your Qt compiler, for example, "Qt 5.13.1 clang 64bits".
7. You should now see the project has opened successfully. On the left panel, you should see two green triangles as shown in Figure 8. Click the green triangle for the compilation. This will run the visualizer.
8. While the visualizer is running, click "Open File" and choose the result.txt file generated by your assignment code. Then, click the "Start" button to start the traffic animation.

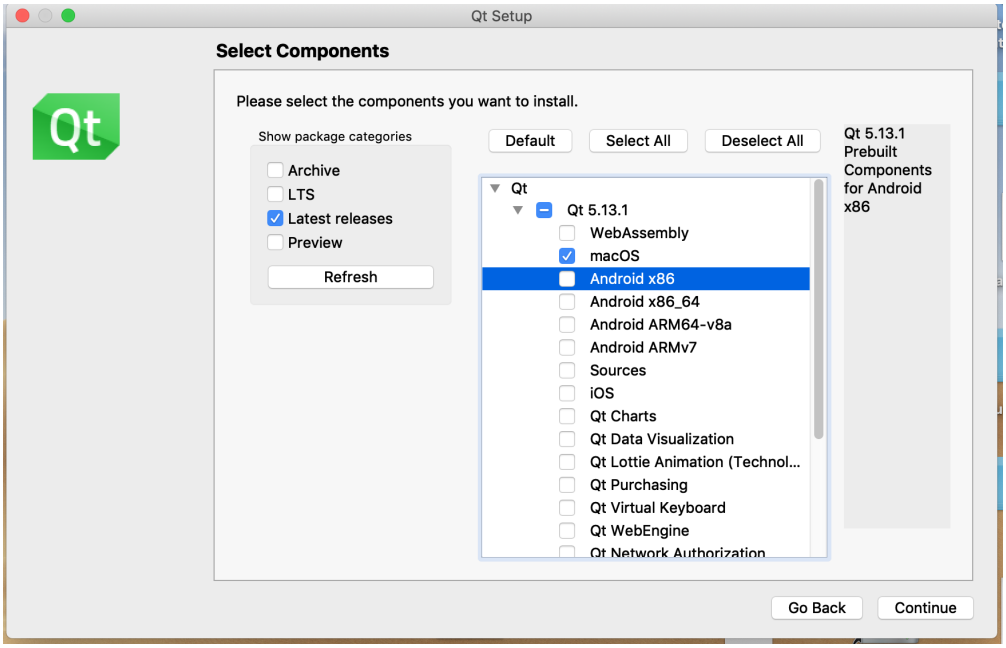

Figure 7: Qt installation.

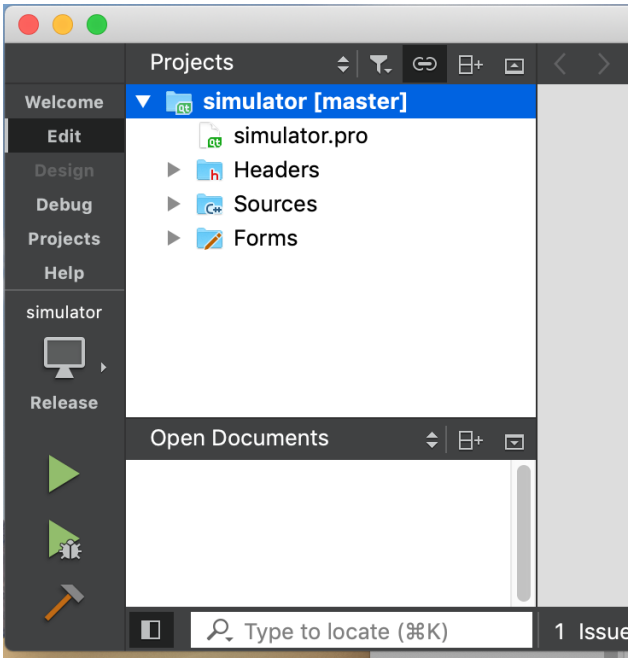

Figure 8: Qt project compilation.

4 How To Complete This Assignment

There are three classes you need to modify/complete. They are TrafficModel, Platoon, and Car. While a platoon is essentially a linked list consisting of many car objects, TrafficModel contains multiple Platoons as a `vector<Platoon>`. Each Platoon object has a pointer to the first car, and a pointer to the last car in the lane. Then, by using the linked list operations, it is possible to navigate between the car objects in a platoon for their position computation.

4.1 Your Tasks

In this assignment, you are free to modify any file in the provided skeleton code. First, have a look at the SimulationControl.cpp file. In the `run` function, it should be straightforward to understand how the simulation works. Basically, in each tick, the operation updates the car position, reads the lane change commands, then writes the result to the output file.

In the provided skeleton code, the `update` function in TrafficModel is left empty for you to complete. This function is responsible for updating the position and state of each car in every tick. Basically, your main task is to implement this function, so that the car positions and state are computed correctly in each tick, according to the movement rules. This will require you to add more methods and attributes in the Platoon and Car classes.

Since the skeleton code is only a rapid prototype of this assignment, you may want to improve the skeleton code in your own coding style. In fact, you can totally ignore the skeleton code and build your own program from scratch. Additionally, you may decide to just use a subset of the skeleton code, or extend the skeleton code. However you decide to modify the skeleton code, **you should make sure that the car objects in a platoon are implemented as a doubly or singly linked list, and that the movement rules are programmed based on the linked list operations** (which you will need to implement).

4.2 Marking Criteria

The marking will be done via examination of the output file (result.txt) generated by your program. More precisely, there will be a set of input.txt files for different scenarios, and each time that your program correctly generates the output result.txt file, some marks will be given. Also, there will be marks on the linked list implementation. **Note that, the Qt visualisation is only an option for testing. Alternatively, you can just inspect the result.txt file directly without the Qt visualizer.** Furthermore, you can assume that **the input file is always provided with the correct format** as shown in Figure 2. You must make sure that the generated output file is always in the correct format as shown in Figure 3.

The total mark for this assignment is 20 with a distribution of 15:5, where 15 marks for automated test case marking, and 5 marks for the code quality inspection. Details are as follows:

1. Test cases (15 marks): the correctness is examined by checking the generated output file. For each case you generate incorrect result, some marks are deduced. **A few test cases and their expected answer will be available on Canvas.** Note that, 5

marks will be deduced if linked-lists are not implemented/used properly, regardless of passing the test cases.

2. Code quality inspection (5 marks):

   - MVC design (2 mark): the skeleton code is already in MVC design. If you complete assignment without breaking the design pattern, you will get these marks.
   - Style (1 mark): consistency on comments and indentation.
   - Efficiency (1 mark): deduction if there are any unnecessary loops.
   - Destructor (1 mark): memory needs to be released appropriately.

4.3 How to submit your code

This assignment deadline is **October 23 11:59 pm**. Submit a zip file, which contains:

1. A zip file of all the source files needed to compile your code.
2. Coversheet: this should include your full name, id, upi, and your sign for declaration of originality.

Please double check your submission, because if your code fails to compile (with the 'make' command), then you will receive ZERO marks for this assignment. There will be no exceptions if you accidentally submitted the wrong files, regardless if you can prove you did not modify them since the deadline. No exceptions.

4.4 Academic Honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help. Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using software similarity tools. Code detected for suspicious similarity will result in an Investigative Meeting and will be forwarded to the Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never give another person your code.
  - Never put your code in a public place (e.g., forum, git, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

4.5 Late submissions

Late submissions incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)

In case if the assignment is extended, then the late submission penalty applies identically to the extension date.