RYERSON UNIVERSITY

Faculty of Engineering, Architecture and Science

Department of Electrical and Computer Engineering

Course Number	**COE 892**
Course Title	**Distributed Cloud Computing**
Semester/Year	**Winter 2022**
Instructor	**Dr. Muhammad Jaseemuddin**

Lab/Tutorial Report NO.	01

Report Title	Lab 1 Report

Section No.	01
Group No.	
Submission Date	2022-02-08
Due Date	2022-02-08

Name	Student ID	Signature*
Darien Lee	500868176	DL

(Note: remove the first 4 digits from your student ID)

**By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work <u>will result in an investigation of Academic Misconduct and may result in a "0" on the work,</u> an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at:*
http://www.ryerson.ca/senate/policies/pol60.pdf.

Introduction:

In this lab, we are aiming to access instructions stored in a public API which will determine the instruction set for each rover. The rovers will move and perform actions depending on the instructions from the aforementioned API. The outputs are text files that represent the paths taken by each rover. In the second part of the lab, the dig function will also be implemented which will go through a “disarming” process. In both parts, the programs will be timed and compared, run sequentially and concurrently.

Implementation:

Part 1:

A *Rover* class was implemented to contain the necessary methods and other variables needed to output the resulting path file. The *Rover* class was implemented to simply receive 1 argument (int rover_number) and the methods defined within the class would run itself (executed in the __*init*__ of the Rover class) to produce the “*path_i.txt*” file for that rover. This was to make a cleaner execution when initialized later in the *main* function.

Some initial states of the rover are defined first in the __*init*__ such as the *self.location = [0,0], self.rover_number = rover_number* and *self.direction = ‘S’*.

The first method executed in the *Rover* class is the “*get_rover_moves*” method which connects to the public API, grabs the necessary data, parses it and stores it as a string “*rover_moves*” for later use.

Following that, the *create_board* method is executed and that data is assigned to the *self.board_list*. The *create_board* method creates a list of lists populated with a default, untraveled path represented by “*0*”s in each sublist. The first value of the first list is preset to “*” as that is the starting point (0,0) of the rover and thus is assumed to be already traveled. A 15x15 board is chosen as it fits the scope of the move sets of the rovers.

The final method directly executed by __*init*__ is the *start_rover* method which will feed each move set instruction into the *set_direction* method. The *set_direction* method functions to determine the appropriate maneuver depending on the instruction received. The set_direction not only updates the *self.direction* declared in __*init*__ to keep track of its orientation but also handles the “M”(move) and “D”(dig, exists but is not implemented in this part) instructions. If the instruction is a “*M*”, the *set_location* method will be called to increment/decrement *self.location* to keep track of the rover’s location and also update the *self.board_list* with the newly traveled track represented by “*”. After the instructions are run to completion, the *write_file* method is called to read the values from the *self.board_list* list and output the results into a “*path_i.txt*”, where “*i*” is the rover number.

The *main* function (outside of the Rover class) simply uses a loop to initialize all 10 rovers which will automatically output the text files. Here, the time function is called to measure the computation time off running each rover script sequentially (through a loop). The multithreading is also implemented in the *main* function so that all 10 threads will be created and run concurrently.

My measured computation time for sequential was about *0.4723 seconds*. My measured computation time for multithreading was about *0.0985 seconds*. The difference was about 0.3738 seconds. It is evident that the multithreading method is significantly faster and more efficient given the necessary resources to do so.

<u>Part 2:</u>

A *Dig* class was implemented to handle the dig operation for a rover. As the lab manual did not state to implement the dig operation into the code from part 1 but to write a new program for this part, the dig operation was not implemented in a form that could be directly imported into the part 1 code. Some small alterations could be made to make this possible but this was assumed to not be within the scope of the lab.

The *Dig* class receives 1 argument, the *mine_number* for initialization. Similar to the *Rover* class implementation, the class will run its contained methods until the condition of *self.disarmed_flag* is set to *True*, indicating that a valid key was found and the mine was disarmed.

There are 4 methods within the Dig class: *get_mine_serial, get_temporary_mine_key, sha256_hash, isValid, and start_disarm.* Each method serves one step in the disarming process of the mine as described in the manual.

The *get_mine_serial* method opens and reads the contents of the "*mines.txt"* file. It then searches for the serial number corresponding to the *mine_number*. Once found, the serial number is stored in the *self.mine_serial* variable for later use. This method is only run once at the beginning.

The *get_temporary_mine_key* method simply increments the *self.pin* variable within the class. Initially, a random integer generator function was used but this would result in inconsistent timing comparisons when comparing between the sequential and threading versions of the class. As such, a simple increment to the predefined variable within the class known as *self.pin* was used to simulate "brute force" for finding a pin. The *self.pin* is concatenated with the *self.mine_serial* to form the temporary mine key stored in the *self.key* variable.

The *sha256_hash* method simply uses the sha256 function from the library *hashlib* on the *self.key* that is stored in a *self.hash* variable.

The *isValid* method checks the first character of *self.hash* and if that character is *0* then we have found a valid key and the *self.disarmed_flag* is set to *True*.

The *start_disarm* method simply runs the *get_temporary_mine_key, sha256_hash and isValid* methods until a valid key is found (while loop check the *self.disarmed_flag*).

The sequential and multithreading versions of this part are implemented the same way as in part 1.

My measured computation time for sequential was about *0.00198 seconds*. My measured computation time for multithreading was about *0.00300 seconds*. The difference was about 0.00102 seconds. In this case, the sequential version ran faster than the multithreading as the program was not very computational expensive so the process of creating and joining the threads were more expensive than the actual operations themselves.

Results:

Name		Last Modified	File size
..		seconds ago	
COE 892 - Lab 1 - Part 1 - Threading.ipynb	Running	11 minutes ago	8.31 kB
COE 892 - Lab 1 - Part 1 Sequential.ipynb	Running	13 minutes ago	7.41 kB
COE 892 - Lab 1 - Part 2 - Sequential.ipynb	Running	a minute ago	4.31 kB
COE 892 - Lab 1 - Part 2 - Threading.ipynb	Running	seconds ago	4.25 kB
Lab1.ipynb	Running	2 days ago	134 kB
mines.txt		4 hours ago	64 B
path_1.txt		12 minutes ago	255 B
path_10.txt		12 minutes ago	255 B
path_2.txt		12 minutes ago	255 B
path_3.txt		12 minutes ago	255 B
path_4.txt		12 minutes ago	255 B
path_5.txt		12 minutes ago	255 B
path_6.txt		12 minutes ago	255 B
path_7.txt		12 minutes ago	255 B
path_8.txt		12 minutes ago	255 B
path_9.txt		12 minutes ago	255 B

Figure 1: Depicts the path_i texts created and the mines.txt

jupyter path_1.txt

File Edit View Languag

```
****00000000000
*0**00000000000
*0***0000000000
*****0000000000
**0*00000000000
****00000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
000000000000000
```

Figure 2: Depicts the output from Rover 1 with its path_1.txt. Other path_i texts follow the same format

```
The computation time was:  0.47231507301330566 seconds
```

Figure 3: Timing of the sequential rover program

```
The computation time was:  0.09849953651428223 seconds
```

Figure 4: Timing of the threading rover program

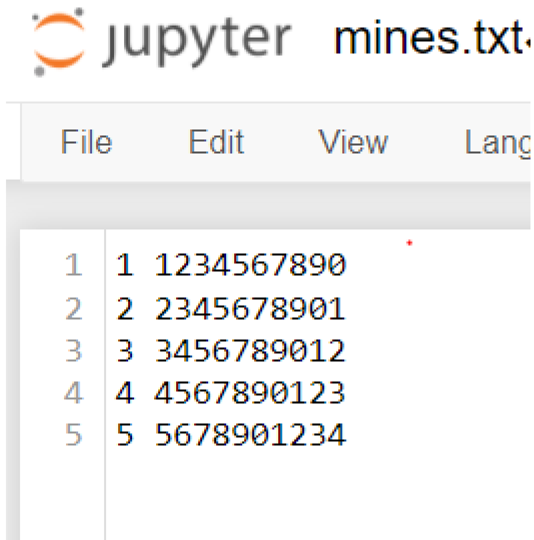

Figure 5: Depicts the contents of the mines.txt file

```
The computation time was:  0.0019774436950683594 seconds
```

Figure 6: Timing of the sequential dig program

```
The computation time was:  0.0030035972595214844 seconds
```

Figure 7: Timing of the threading dig program

<u>Conclusion:</u>

In this lab, we have successfully retrieved data from a public api and used that data to make a rover program. We have also compared the sequential and threading versions of each rover and dig programs to analyze the differences. As a result, we can see that although multithreading is expected to be faster, there are some cases where the threading processes themselves cost more than the actual program and may not be ideal. Another difference that I noticed but did not record was that the Rover programs which required accessing the public API took more time to run at home than they did at the University. This may be due to many factors such as internet speed, network traffic, proximity to servers, and many other factors.