



Course Title:	
Course Number:	
Semester/Year (e.g.F2016)	

Instructor:	
--------------------	--

<i>Assignment/Lab Number:</i>	
<i>Assignment/Lab Title:</i>	

<i>Submission Date:</i>	
<i>Due Date:</i>	

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, 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/current/pol60.pdf>

COE 892 Lab 1 Assignment Explanation

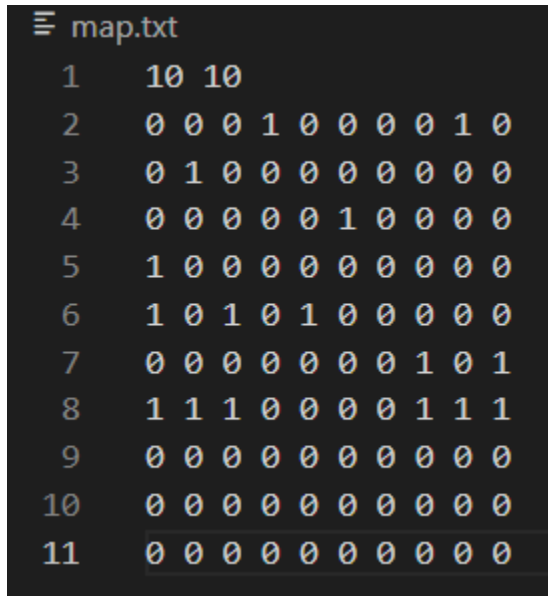
PLEASE NOTE: Part 1 and Part 2 of this lab assignment have been split into two separate python files. To run and test the output for part 1, please run the lab1Part1.py file. To run and test the output for part 2, please run the lab1Part2.py file.

Introduction

The main purpose of this lab is to use various python programming modules and libraries to process the data that is sent to several land mine detector robots. This lab assignment is split into two parts. Part 1 asks to take in the commands for each robot as input and outputs a drawn path that the rover takes on the land space. Part 2 on the other hand asks to create a sha256 hashing algorithm that is used to simulate the digging and the disarming process of mines for rover 1 when rover 1 comes into contact with a mine. The specific details regarding the thought processes for how I programmed parts 1 and 2 for this assignment, can be noted below.

Part 1: Drawing Rover Paths

Firstly, for part 1 as stated in the introduction, I am asked to read in the commands of each robot as input and use those commands to draw out the path that the robot is expected to take on the robot's land space with asterisk ("*") characters. This land space is stored inside of a file called "map.txt". The format of the map.txt is as follows: The first line of the text file contains the length and width dimensions of the land space (i.e., 4x6, 5x5, 10x10 etc.). the remainder of the lines in the text file shows a 2D matrix that indicates the spots where various mines are located on the land space. The screenshot below shows what it should look like if formatted correctly:



```
≡ map.txt
1    10 10
2    0 0 0 1 0 0 0 0 1 0
3    0 1 0 0 0 0 0 0 0 0
4    0 0 0 0 0 1 0 0 0 0
5    1 0 0 0 0 0 0 0 0 0
6    1 0 1 0 1 0 0 0 0 0
7    0 0 0 0 0 0 0 1 0 1
8    1 1 1 0 0 0 0 1 1 1
9    0 0 0 0 0 0 0 0 0 0
10   0 0 0 0 0 0 0 0 0 0
11   0 0 0 0 0 0 0 0 0 0
```

Figure 1: My map.txt file for Lab 1

A "1" on the map, indicates that a mine is buried in that location of the map while a "0" on the other hand indicates that nothing is buried in that location of the map. For the rover commands of each rover, those be accessed by visiting the following web URL link: <https://coe892.reev.dev/lab1/rover/:number> – where the number field relates to the corresponding

rover number. For example, if we replace the number field with 1 and subsequently click on the URL, it will redirect the user to a JSON file (which can be freely viewed in one's web browser of course) that contains the command set for rover 1. Below is a screenshot of what this JSON file looks like:

```
{"result":true,"data":{"moves":"MMMMRMLRRRLRMLMRMLMMMLMMRMMLDMLMMMMMLRLLRDMMLDMLRDMRLMRMRMRRLRLLRMDRMRDLMDLM"}}
```

Figure 2: JSON File Data for Rover 1

Since the move set data for rover 1 is stored inside of a JSON file, the first step for implementing part 1 of the assignment was to create a function which takes the JSON file data for each rover and to store that data in a variable as a dictionary. This is because JSON data can only be represented as dictionary structures in the Python programming language. Below is a screenshot of my implementation for this function:

```
7   #Converts Json file into python dictionary
8   def httpRip(rovNum):
9       url = "https://coe892.reev.dev/lab1/rover/" + str(rovNum)
10      fp = urllib.request.urlopen(url)
11      byteArr = fp.read()
12      dict_string = byteArr.decode("utf-8")
13      fp.close()
14      rover_dict = json.loads(dict_string)
15      return rover_dict
16
```

Figure 3: Extract contents of JSON file

As you can see above, I was able to open up and read the data from the URL via the urllib module in Python 3. More specifically, I used the urllib.request.urlopen function to access said data. Then, since the urlopen function returns a pointer to a file object, in order to convert the url data into a dictionary, I first have to take the file object and convert it into a byte array with fp.read(). Then I convert the byte array into a string by using the decode method. Lastly, after I close the file object, I use json.loads() to finally convert the string to a dictionary.

For the next step in my process, since the newly converted dictionary from the JSON file also has another dictionary inside of it, I decided to create another function that extracted and returned that inner dictionary by using the output of the previous function. It simply iterates over the dictionary that was returned from the previous function and if it detects that one of the keys is of data type 'dict' (for dictionary), then it returns the inner dictionary as the result. Below is a screenshot of my implementation for this other function.

```
#Extracts the {moves: MMMMR...} inner dictionary out of the main outer dictionary of the JSON file
def command_seperation(rovD):
    for key in rovD:
        if type(rovD[key]) is dict:
            command_dict = rovD[key]
            break
    return command_dict
```

Figure 4: Extracting the inner dictionary that contains the move set data from the main JSON dictionary.

Now, when this inner dictionary is returned by the function above, the dictionary is in the following form (the following is a random example): {"moves": "MMMMMLRM"}. Because of this, since we have access to the data of 10 rovers total, it will be hard to keep track of which move set corresponds to which rover. Therefore, I decided to write another function that again, takes the dictionary output of the function noted in the screenshot above as input and returns a modified version of that dictionary that changes the "moves" key component of that dictionary to the rover number (which is a string) that is associated to a given move set. For example, instead of {"moves": "MMMMMLRM"}, it would be changed to {"1": "MMMMMLRM"}, which indicates that the move set "MMMMMLRM" is the move set of rover 1. In doing so, it helps keeps the move sets of our 10 rovers more organized and easily identifiable as a result. A screenshot of this function's implementation is shown below.

```
#Takes the 'moves' string from the inner moves dictionary and changes it to the associated rover number
def extract_movements(CD,n):
    for key in CD:
        if key == "moves":
            CD[str(n)] = CD[key]
            del CD[key]
            break
    new_CD = CD
    return new_CD
```

Figure 5: Changes the 'moves' key from the inner dictionary to the corresponding rover number of a given move set. The rover number is a string.

The next step in my process was that I then took the output of the previous function noted above as input for another function and extracted the move set string (i.e., “MMMMMLRM”) from it and stored it in an array. The screenshot below shows the implementation for this function.

```
# Takes the sequence of movements for each rover from the moves dictionary and converts it into a multi-character string in a single element list
# similar to something that is like the following format: [MMMMMDRMD]
def get_command_list(new_CD, i):
    for key in new_CD:
        if int(key) == i:
            commandList = list(new_CD.values())
            break
    return commandList
```

Figure 6: Extract the moves string and convert it to a list.

Next, I took the moves set string array output from the previous function and created another function that converts the moves set string array back to a regular string and takes that regular string and converts it into an array of characters (i.e. [M, M, M, R, L]). Below is a screenshot of the implementation.

```
# Takes the output from the get_command_list function, converts the single-element list into a regular string and takes the string and converts
# it into a character array. In this character array, each character represents a rover command
def commandChars(comList):
    commands = ''.join(comList)
    commandChars = list(commands)
    return commandChars
```

Figure 7: Converting the move set string list into a string of characters.

The next function that I wrote takes the matrix data from the “map.txt” file as input, converts it and returns it as a 2D python array. Below is a screenshot of the function’s implementation.

```
# Takes the data from the map.txt file and converts into a 2D list
def create_matrix(file):
    landSpace = []
    count = 0
    with open(file, "r") as f:
        lineList = f.readlines()
        for line in lineList:
            count = count + 1
            if count >= 2:
                mat = [item.strip() for item in line.split()]
                landSpace.append(mat)
    return landSpace
```

Figure 8: Converts matrix data from map.txt into a 2D array and returns said 2D array.

Before I continue on to the next step in my process, I would also like to state that the array of move characters that were returned by the function noted above, consist of four different move types. The “M” move character or command, tells the rover to move forward to the next space on the map with respect to the direction that the rover is currently facing, and the “D” command tells the rover to dig a hole in the ground at its current location in the land space. The “L” command tells

the rover to turn left. This means that if the rover is initially starting at the top left corner of the map and is initially facing in the south direction. If the rover receives the command to turn left, the rover will then change its direction to face east. If it receives another “L” command, the rover will face north. Essentially, if the rover keeps receiving L commands from the moves list, the rover will always counterclockwise assuming that initially, before the rover receives any commands, it is facing in the south direction. Vice versa, If the rover is initially facing south and continuously receives “R” commands, it will always rotate clockwise. With this explanation in mind, After I created the function that converts the move set string list into a list of characters, I created a function that determines the direction that the rover is currently facing while it traverses the land space. This function is also used to change the direction based on the rover’s current direction and whether the rover receives an “L” command or an “R” command. The screenshot for this function’s implementation is noted below:

```
# Determines the current direction that the rover is facing, given the command that is sent to the rover
def directionChange(command,currentDirection):
    # LEFT COMMAND CASES
    if command == "L" and currentDirection == "S":
        currentDirection = "E"
    elif command == "L" and currentDirection == "E":
        currentDirection = "N"
    elif command == "L" and currentDirection == "N":
        currentDirection = "W"
    elif command == "L" and currentDirection == "W":
        currentDirection = "S"
    # RIGHT COMMAND CASES
    elif command == "R" and currentDirection == "S":
        currentDirection = "W"
    elif command == "R" and currentDirection == "W":
        currentDirection = "N"
    elif command == "R" and currentDirection == "N":
        currentDirection = "E"
    elif command == "R" and currentDirection == "E":
        currentDirection = "S"
    return currentDirection
```

Figure 9: Function that determines the direction that a rover is currently facing given the command that it receives. It takes the rover command and the rover’s current direction as input string values.

The next function in my code which I wrote is of great length, but it essentially the primary function that is used to draw the full path of the rover as it traverses through the entire land. To start off the function, it takes two input values, which consist of the command list that was obtained from the commandChars function (see figure 7) and the 2D array representation of the land space that was obtained from the create matrix function (see figure 8). First a secondary copy of the 2D land space array is stored in a variable called pathMat. This pathMat variable is then used to store and keep a record of the drawn path for the rover. We then initialize the starting coordinate position for the rover to be (0,0) and the starting direction that the rover is facing is initially set to south or the “S” character. We then iterate through the list of commands for the rover via a for-loop and check if the rover accidentally goes out of bounds (for error-handling purposes). If it does, we exit out of the for-loop immediately and return the pathMat variable which contains the path that has been drawn with asterisks so far. If the rover doesn’t go out of bounds, we compute series of if-elif-else statements that check if the rover receives an “M” command while also verifying what direction that the rover is currently facing. This is important because the rover’s current direction

dictates how the rover's current position will change. For example, if the rover is moving south, the x-coordinate value would increase but the y-coordinate value would remain constant. If the rover is moving north, the x-coordinate would decrease. If the rover is moving west, the y-coordinate would increase and if it were moving east, the y-coordinate would decrease. So, given the rover's current direction and the fact that the rover has received a move command, if the rover is located in a spot where it is within the boundaries of the map and it can realistically move forward to another space on the map, the function first marks its current location with an asterisk symbol ("*") and then it would change the current xy-coordinate location to the new xy-coordinate location that it should move to on the map. If the rover accidentally drives over a land mine that is buried in one of the spots on the map (indicated by a "1") and does not dig it out, then the rover would explode, and the function would first mark the spot where it got destroyed with an asterisk and then break out of the loop and return the path that has been drawn out for the rover so far. The program would then move on to drawing the path for the next rover in the series of 10 rovers. If the rover does not do this and drives over a spot that does not have mine buried underground (indicated by a "0"), then the rover is safe from exploding and the rover would be able to proceed with processing the next command in the command list for the rover. If the command received by the rover is a turning command such as "L" or "R", then the rover would simply change its current direction by using the direction change function that was written previously (see figure 9). Lastly, if the rover receives a dig command, the rover digs a hole at the spot where the rover is currently located. The function should also mark that digging spot with an asterisk regardless of whether that particular spot contains a mine buried underneath it or not. At end of the function, it returns the complete path that the rover has taken. This path is stored in the variable pathMat. Below are some screenshots of this function's implementation.

```

86 #Creates a 2D list that contains the rover's path on the land space map
87 def createPath(landMat, commandMat):
88     pathMat = copy.deepcopy(landMat)
89     pathRows = len(pathMat)
90     pathCols = len(pathMat[0])
91     currentFacing = "S"
92     x = 0
93     y = 0
94     for command in commandMat:
95         if x > pathRows - 1 or y > pathCols - 1 or x < 0 or y < 0: #Check if the rover goes out of bounds for the land space
96             break
97         else:
98             # Cases where command is M
99             if currentFacing == "S" and command == "M":
100                 if pathMat[x][y] == "1": #Rover is on a mine
101                     pathMat[x][y] = "*"
102                     break
103                 elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
104                     pathMat[x][y] = "*"
105                     if x < pathRows - 1:
106                         x = x + 1
107             elif currentFacing == "N" and command == "M":
108                 if pathMat[x][y] == "1": #Rover is on a mine
109                     pathMat[x][y] = "*"
110                     break
111                 elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
112                     pathMat[x][y] = "*"

```

Figure 10: Function that is used to draw out the complete path that the rover takes on the land space (Part A).

```

111         elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
112             pathMat[x][y] = "*"
113         if x > 0:
114             x = x - 1
115         elif currentFacing == "E" and command == "M":
116             if pathMat[x][y] == "1": #Rover is on a mine
117                 pathMat[x][y] = "*"
118                 break
119             elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
120                 pathMat[x][y] = "*"
121             if y < pathCols - 1:
122                 y = y + 1
123         elif currentFacing == "W" and command == "M":
124             if pathMat[x][y] == "1": #Rover is on a mine
125                 pathMat[x][y] = "*"
126                 break
127             elif pathMat[x][y] == "0": # you can move and the rover is not over a mine
128                 pathMat[x][y] = "*"
129             if y > 0:
130                 y = y - 1
131         # If command is D, place * whether 1 or 0
132         elif command == "D":
133             pathMat[x][y] = "*"
134         # If command is L or R, change direction of rover
135         elif command == "L" or command == "R":
136             currentFacing = directionChange(command,currentFacing)
137     return pathMat

```

Figure 11: Function that is used to draw out the complete path that the rover takes on the land space (Part B).

The last function that I created takes the completely drawn-out path matrix output from the function noted above and writes that output into a new text file that is labeled as “path_i.txt” where “i” refers to the rover number. So for example, if the file is called “path_1.txt” then this file would contain the completely drawn-out path for rover 1. This function marks the end of my path drawing algorithm for a rover. The screenshot for my implementation of this function is noted below:

```

#Takes the current rover's complete path on the land space path and writes that data to a new txt file
def writeMatToFile(pathMatrix,index):
    if os.path.exists("path_" + str(index) + ".txt"):
        os.remove("path_" + str(index) + ".txt")

    fileName = "path_"+str(index)+".txt"
    roverPath = open("path_"+str(index)+".txt","x")
    roverPath.close()
    with open(fileName,"w") as pathFile:
        for row in pathMatrix:
            pathFile.write(' '.join([str(a) for a in row])+ '\n')

```

Figure 12: Function that writes the drawn path output of the path drawing function to a text file

Now that my algorithm is complete, I then decided to put all of the functions that are a part of my path drawing algorithm inside of one outer function called algo(), which takes the number value of the current rover as input. The reason why I did this is to help make the algorithm compatible

with Python 3's threading library and to help test my algorithm's overall performance as well. Therefore, in order to use this algorithm with a multithreading approach, I created a for loop that runs a total 10 times (one iteration per rover) and each index of the for loop corresponds to a rover's id number (index 1 is for rover 1, index 2 is rover 2 and so on). I called my algo() function in this for loop and I also put lines of code in the loop that create thread objects which execute my path drawing algorithm function algo(). This means that in my program I basically created a thread object that executed the algorithm for each rover. In doing so, I have basically created a total of 10 threads where each thread executes the same algo() function in a multithreaded manner. I also used the performance counter function from python 3's timer module to calculate the total amount of time that this multithreading approach takes for my algorithm as well. I also created a second for-loop that runs 10 times which is used to calculate the sequential performance of my algorithm for all 10 rovers. So basically for the sequential approach, I simply called my algo() function in this secondary for loop that runs 10 times and the current index of this secondary for loop is fed into my algo() function as input to indicate that the index corresponds to a single rover's id number. I also used the performance counter function from the time module to calculate the amount of time that this sequential approach takes for my algorithm as well. Below is a screenshot of my implementations for these approaches below:

```

153  # Executes the path drawing algorithm for rovers 1 to 10
154  def algo(i):
155      rovDict = httpRip(i)
156      comm_dict = command_seperation(rovDict)
157      ncd = extract_movements(comm_dict,i)
158      commList = get_command_list(ncd,i)
159      land = create_matrix("map.txt")
160      commandCharList = commandChars(commList)
161      pathMatrix = createPath(land,commandCharList)
162      writeMatToFile(pathMatrix,i)
163      sleep(1)
164
165  st1 = perf_counter() #Start time counter for threaded approach
166  for x in range(1,11):
167      t = Thread(target=algo(x))
168      t.start()
169      t.join()
170  et1 = perf_counter() #End time counter for threaded approach
171
172  print(f'The multithreaded approach took {et1- st1: 0.2f} second(s) to complete.')
173
174  st2 = perf_counter() #Start time counter for sequential approach
175  for y in range(1,11):
176      algo(y)
177  et2 = perf_counter() #End time counter for sequential approach
178
179  print(f'The sequential approach took {et2- st2: 0.2f} second(s) to complete.')

```

Figure 13: Algorithm function and Algorithm Performance Calculations for Multithreading and Non-multithreading approaches

The multithreaded approach took 10.93 second(s) to complete.
The sequential approach took 10.75 second(s) to complete.

Figure 14: The Performance Times for Multithreading and Sequential Approaches

Part 2 – Digging Mines with Rover 1

For part 2 of this lab assignment, we are asked to consider the commands of only rover 1. More specifically, we want to consider the digging procedure for just rover 1 (**please note that in part 2, the lab manual only considers the digging procedure of rover 1. For this reason, only the command list for rover 1 will be mentioned in this section of the report**). The way that the digging process works for rover 1 is that when rover receives a command dig for a mine while it is currently on top of a mine, it initiates the complete digging and disarming process for that mine. The way that this digging and disarming procedure works is that first when rover 1 hits an arbitrary mine on the land space and receives a “D” command, it searches through a file called “mines.txt” to retrieve the hit mine’s serial number which I decided to represent as a universal unique identifier or UUID for short. After retrieving the serial number that is associated with the mine that was hit by rover 1, the program then selects a randomly generated pin value and concatenates said pin value to the mine serial number to create an encrypted temporary mine key. After the creation of this key, the program then uses a sha256 hashing function to hash and essentially decrypt the encrypted mine key data. If the resultant decrypted key data has six leading zeros (i.e. “000000”), then this means that the selected pin is a valid pin and that rover can successfully disarm the mine that it came into contact with. If the selected pin in any case is not valid, then a different randomly generated pin must be selected and it will repeat the hashing process again with this new pin. As a result, this process of randomly generating and concatenating various pins for hashing a valid pin value essentially continues indefinitely until a valid pin hash with six leading zeros is produced. My python 3 coding process for part 2 of this assignment is 95% similar to my part 1 coding process with the key differences being that a function that was added to simulate the digging and disarming process for the mines that rover 1 comes across as well as another function that takes and reads in the data of a text file, stores the contents of said text file in a 1D array, and returns this array as the final result. Due to this high similarity factor, I will only be highlighting the lines of code and the functions that were newly added to my part 1 code as part of my part 2 code.

First, I created a new python file that would store all the functions, loops, and other processes for part 2. Then in this new python file, first I copy and pasted all the code from my part 1 file into my part 2 file. Next, I removed the for-loop structure from my algorithm and changed the rover number inputs for all of my functions (where applicable) to 1 since we only consider the actions of rover 1 only for part 2. Below is a screenshot of these changes:

```

rovDict = httpRip(1)
comm_dict = command_seperation(rovDict)
ncd = extract_movements(comm_dict,1)
commList = get_command_list(ncd,1)
land = create_matrix("map.txt")
commandCharList = commandChars(commList)
pathMatrix = createPath(land,commandCharList)
writeMatToFile(pathMatrix)

```

Figure 15: Modified algorithm that only accounts for rover 1.

While I also kept the function which writes the complete path data to a file, I slightly modified it for part 2 with the simple change being that instead of the function creating a file called “path_1.txt”, it creates a file called output.txt. The reason why I did this is primarily due to testing purposes and making sure that my code for part 2 works as it should be expected to while meeting all the requirements that are stated in part 2. Below is a screenshot of this slight modification:

```

176 ✓ def writeMatToFile(pathMatrix):
177 ✓     if os.path.exists("output.txt"):
178         os.remove("output.txt")
179
180         fileName = "output.txt"
181         roverPath = open("output.txt","x")
182         roverPath.close()
183 ✓     with open(fileName,"w") as pathFile:
184 ✓         for row in pathMatrix:
185             pathFile.write(' '.join([str(a) for a in row])+ '\n')

```

Figure 16: Write path to a file function is modified to write to a file called output.txt instead of path_1.txt. Executes the exact same action as the original function.

Next, I created a function that is used to read, store, and return the the mine serial number data from the mines.txt file as 1D array. The formatting for my “mines.txt” file is as follows: Each line consists of the xy-coordinate location for a mine on my map.txt followed by a space which is then followed by the serial number for the mine at that xy-coordinate location. Below is a screenshot example of what this format (if done correctly) should look like:

```

≡ mines.txt
1    03 1cc83792-a35a-11ed-a8fc-0242ac120002
2    08 2ab41d3a-a35a-11ed-a8fc-0242ac120002
3    11 3aa5570e-a35a-11ed-a8fc-0242ac120002
4    25 5390e6fc-a35a-11ed-a8fc-0242ac120002
5    30 e2948ab0-a35b-11ed-a8fc-0242ac120002
6    40 fc441cb4-a35b-11ed-a8fc-0242ac120002
7    42 072e2af2-a35c-11ed-a8fc-0242ac120002
8    44 92d9912c-a35c-11ed-a8fc-0242ac120002
9    57 9cb4f5f6-a35c-11ed-a8fc-0242ac120002
10   59 a9fb1f60-a35c-11ed-a8fc-0242ac120002
11   60 83393dc4-a35e-11ed-a8fc-0242ac120002
12   61 8d012f24-a35e-11ed-a8fc-0242ac120002
13   62 93f37490-a35e-11ed-a8fc-0242ac120002
14   67 9c7b7bc6-a35e-11ed-a8fc-0242ac120002
15   68 cec5575e-a35f-11ed-a8fc-0242ac120002
16   69 d9e3d386-a35f-11ed-a8fc-0242ac120002

```

Figure 17: The contents of my mines.txt file.

As you can see in the above screenshot for example, the first line is written as 03 and then a space and then followed by the space, the rest of the first line consists of the serial number for the mine that is located at 03. 03 in this context refers to the mine that is located at x-coordinate 0 and y-coordinate 3. Therefore, in the context of the mines file, the first line can be read as the following: The mine located x-coordinate location 0 and y-coordinate location 3 has the mine serial number of 1cc83792-a35a-11ed-a8fc-0242ac120002. Below is a screenshot of my implementation of the function that reads in the data of the mines.txt file and returns it as a 1D python array:

```
def ripMineData(file): #Reads the mines.txt file, then stores and returns the mines.txt file's information as a 1D list
    mineData = []
    with open(file,"r") as f:
        lineList = f.readlines()
        for line in lineList:
            mineData.append(line.strip())
    return mineData
```

Figure 18: Function that reads in the information from mines.txt and returns it as a 1D array.

The next function that I created for part 2 of this assignment was the primary digging and disarming function that will be used to simulate the digging and disarming of mines for rover 1. Below is a screenshot of this function's implementation:

```
95 def digProces(id): #Digging and disarming procedure for the rover 1 when rover 1 hits a mine and receives a 'D' command
96     validPin = False
97     pin = 0
98     while not validPin:
99         mineKey = str(pin) + id
100         hashedValue = sha256(mineKey.encode('utf-8')).hexdigest()
101         if hashedValue[0:6] == "000000":
102             validPin = True
103         else:
104             pin = pin + 1
105     return validPin
```

Figure 19: Digging and Disarming function for rover 1

As it was mentioned earlier in this report, whenever rover 1 hits an arbitrary mine on the map and receives a 'D' command, the program searches through the mines.txt file and retrieves the hit mine object's correct associated serial number (which can be verified by checking the mine's associated location coordinates [also stored in the mines.txt file]). After retrieving the mine's associated serial number, the program then calls the function noted in the screenshot above by using the serial number as an input parameter. First a boolean variable is declared and initialized as false. Then, an initial pin value of zero is established. Then the function executes a while loop that concatenates this initial pin value to the mine serial number to create an encrypted temporary mine key. Continuing on in the loop, the encrypted mine key is then decoded or hashed by a sha256 hash function which can be imported from python 3's hashlib module. If the decoded hashed value that is returned by the sha256 function contains six leading zeros, then the pin that was chosen to create the temporary mine key was a valid pin number and the boolean variable that was declared and initialized at the start of the function is then changed to true. Otherwise, a new, different pin is generated and concatenated with the serial number in order to create another new and different mine key and repeat the hashing or decoding process again. Essentially, this purpose of this function is to continuously generate and hash various temporary mine keys until, one of the mine keys outputs a hash value with six leading zeros. If this case occurs, the pin value used to create the mine key is a valid pin and the mine is successfully disarmed.

The last step that I took for my part 2 algorithm of the lab 1 assignment is that I took the createPath function (see figures 10 and 11), and modified one of the if statements to incorporate the inclusion of using a mines.txt file to locate and verify mine serial numbers and the inclusion of a digging and disarming function which is used to simulate the digging and disarming of mines. The if statement in question which I have modified for part 2 is the if condition that checks whether the rover receives a “D” command while it is on top of a mine object (if the current spot on the map is equal to a 1). Below is a screenshot of the createpath function with the modified if condition:

```
mineMatchcount = 0
mineLoc = ""
disarmed = False
mines = ripMineData("mines.txt")
```

Figure 20: Newly declared and initialized variables to help execute modified createPath function.

```
# If command is D and the rover does NOT hit a mine, write a * at that location
elif command == "D" and pathMat[x][y] == "0":
    pathMat[x][y] = "*"
elif command == "D" and pathMat[x][y] == "1": # If command is D and the rover does hit a mine, initiate the digging process for that mine
    for j in range(len(mines)):
        mineLoc = mines[j][0] + mines[j][1]
        if mineLoc == str(x) + str(y):
            mineMatchcount = mineMatchcount + 1 # Counter which keeps track of the total number of mine serial location matches
            disarmed = digProces(mines[j][3:])
            if disarmed == True:
                print("Mine located at x-coordinate " + mines[j][0] + " and y-coordinate " + mines[j][1] + " has been disarmed successfully!")
                pathMat[x][y] = "#" # If mine is successfully disarmed place a # character at that location
                break
    elif mineMatchcount == 0 and j == len(mines) - 1:
        print("Mine cannot be disarmed. Unable to identify the mine's serial number.")
        pathMat[x][y] = "*" #Error handling scenario where the function is unable to identify the mine's serial number
    else:
        continue
```

Figure 21: Modified createPath function with separated if conditions given that a D command has been received by the rover.

As you can see from the screenshot if the rover moves on top of a mine object and receives a ‘D’ command, first the function iterates through the mine serial data list that is returned by the ripMineData function. It does this to check if the rover’s current location coordinates match the location coordinates for one of the mines from the mines.txt file. If there is an exact match, then the digging and disarming mines function executes immediately. Once the function has obtained a valid pin, the mine is disarmed and the location where the mine is disarmed is then marked with a # sign to indicate that the mine at that location has been disarmed successfully. In the case where the function is unable to identify the serial number of the mine, it will print out an appropriate error message to indicate that the serial number could not be found.

After this step, my algorithm for part 2 was complete. After completing my algorithm, I decided to measure the performance of my algorithm in a multithreaded approach versus a sequential approach. To do this, first I created a function called algo() which contains and executes all of the functions that are associated with the algorithm that I have created for part 2. Then I use python’s performance counter function from the time module to measure the total time that it takes for the algo() function to run. The total time measurement result that I get from this is the performance result for the sequential approach for my algorithm. As for the multithreaded approach for my algorithm, I essentially used python’s threading module and created a thread object which ran a

single instance of my algo() function and I used the performance counter function to calculate the total time that it took this threaded approach to run. The time result that I got from this is the performance result for the threaded approach for my algorithm. Below of is a screenshot of how I implemented both of these approaches as well as my time results for both approaches.

```

194 # Executes the path drawing algorithm for rover 1 which has been modified to incorporate the digging and disarming processes of mines on the
195 def algo(): # land space
196     rovDict = httpRip(1)
197     comm_dict = command_seperation(rovDict)
198     ncd = extract_movements(comm_dict,1)
199     commList = get_command_list(ncd,1)
200     land = create_matrix("map.txt")
201     commandCharList = commandChars(commList)
202     pathMatrix = createPath(land,commandCharList)
203     writeMatToFile(pathMatrix)
204
205     st1 = perf_counter()
206     t = Thread(target=algo())
207     t.start()
208     t.join()
209     et1 = perf_counter()
210     print(f'The multithreaded approach took {et1- st1: 0.2f} second(s) to complete.')
211
212     st2 = perf_counter()
213     algo()
214     et2 = perf_counter()
215     print(f'The sequential approach took {et2- st2: 0.2f} second(s) to complete.')

```

Figure 22: Sequential and Threaded Approaches for Part 2 Algorithm

The multithreaded approach took 0.10 second(s) to complete.
The sequential approach took 0.07 second(s) to complete.

Figure 23: Performance Results for threaded and sequential approaches. Does not verify correctness of digging and disarming mines function

Please note that the performance results indicated in the screenshot above are unable to verify whether the digging and disarming function accurately work as they are expected to. This is because the default command list for rover 1 has more move statements than it has dig statements (See Figure 2 for rover 1's command list). Therefore, in order to test and verify if the digging and disarming function actually works as it is intended to work, then it is highly advised to feed in a "dummy" command list such as ['M','M','M','D','M','D','M','M'] for example in order to verify the digging and disarming function's correctness. The screenshot below shows the result of feeding this example dummy command list as an input parameter to the createPath function.

Mine located at x-coordinate 3 and y-coordinate 0 has been disarmed successfully!
Mine located at x-coordinate 4 and y-coordinate 0 has been disarmed successfully!
The multithreaded approach took 30.36 second(s) to complete.
Mine located at x-coordinate 3 and y-coordinate 0 has been disarmed successfully!
Mine located at x-coordinate 4 and y-coordinate 0 has been disarmed successfully!
The sequential approach took 32.44 second(s) to complete.

Figure 24: Performance results of algorithm if dummy command list [['M','M','M','D','M','D','M','M']] is fed into createPath function as an input parameter.

Conclusion

To draw conclusions, my performance times for the sequential approach and the threaded approach is the same and this is true for both part 1 and part 2. This is to be expected for both of my programs because of threads work conceptually. For the sequential approach of part 1 for this lab assignment, I created an algo function which ran a total of 10 times (one execution per rover). While for the

threaded approach on the other hand I created a thread for each rover that executed the algo() function in a multithreaded fashion. Both these approaches gave the same performance value because a thread object essentially runs a single instance of a program. Therefore, if we create and start a thread that runs the algo function for every single rover, it essentially runs 10 instances of the entire algo function in a sequential manner after each thread is finished running their instance of the function. This yields the same result as running the algo() function in a for loop that executes a total of 10 times. Because of this, both approaches yield similar performance results for part 1. Part 2 follows this exact same logic with the only difference being that its only one rover and because its only one rover, only 1 thread is used for the threaded approach. It also means that the algo() function is only called once due to only running it for one rover as well for both approaches.