Etude 12: Contact Tracing

Alex Lake-Smith 5400306
Mathew Shields 2419874

To run the code

Enter the command:
java ContactTrace *nameofdatafile.txt*

- Where nameofdatafile.txt is the dataset you want to search

You can then execute multiple queries within this one line of code. To quit the program type either quit or press control+c.

MultiThreading

To implement our multithreading program we created a new subclass called MyThread which extends the Thread class. Depending on how many threads were chosen, the data would either be split in half for 2 threads, into 4 parts for 4 threads or run as a whole individual list if only using 1 thread. To do this a for loop is used to iterate over an ArrayList which contains all the traces and splits it into smaller lists. The system then sets a start variable at the moment the thread is started or when the lists are initially split. Once each of the threads have been joined and finished running a finish variable will be set to the system's current time, the runtime is then printed in milliseconds once all threads have been executed.

Testing Threads on different sized datasets

Trace1.txt and Trace2.txt are sourced from the **Blackboard** page under Etude 12
The two bottom trace tests were done on files filled with random numbers which were generated by using the **ct.c file** provided on **Blackboard**

	No Threads (millis)	**1 Thread (millis)**	**2 Threads (millis)**	**4 Threads (millis)**
trace1.txt **10 Traces** Query: ID**: 4** Time**: 11**	0 milliseconds	0 milliseconds	1 milliseconds	0 milliseconds
trace2.txt **9881 Traces** Query: **ID: 74** **Time: 140**	6 milliseconds	5 milliseconds	4 milliseconds	3 milliseconds
Custom Generated File **1,000,000 Traces** Query: **ID: 100** **Time: 77**	54 milliseconds	53 milliseconds	31 milliseconds	16 milliseconds
Custom Generated File **2,000,000 Traces** Query**:** **ID: 199** **Time: 200**	71 milliseconds	69 milliseconds	42 milliseconds	26 milliseconds

Upon analysis we found that using threads often outperforms the non-thread search method. This is because the computer is maximizing it's CPU usage and is able to distribute tasks using a minimal amount of memory. My computer has 4 cores, so when it comes to running commands, it is like I have 4 separate computers which are able to be called within my laptop. Initially my thread method was performing at exceptional speeds but this was because we were not joining them together to ensure that they are incrementally ended and then outputting the time for all threads to run and then finish. The higher number of cores that you run the program on, the more threads the program can handle efficiently, thus reflecting why the results from the table generally start at a higher runtime and as it inches closer to using 4 threads the runtime significantly improves.

I found that it is not worthwhile using threads on a data file as small as the first two trace files to test as the difference between thread and non thread runtimes are so slim that you can't really say that it is much of a performance boost. However, threads become more beneficial when the files become larger and the search space increases. I found that some of the results can be inconsistent. For example, most of the time four threads would be quicker than two threads or two threads quicker than one thread but there may be irregularities in the results such as one big jump or drop in time. I originally was not doing my parallel computing correctly as I was using four different lists for four threads to keep track of the separate results and then merge them, however upon feedback I implemented this in my threads. I also was splitting the lists every time I ran a different query which may have led to more overhead time, therefore I only split the list once and then run all queries that may be left on the same split list. I also had to check if merging the lists results caused any significant time overhead but after printing the time for merge it was 0-1ms in runtime so I thought this was not significant enough to be causing the jumps in time. However, Zhiyi explained to me that these threads are not expected to perform the exact same every time that a new query is passed and Java may be performing some extra overhead which may lead to different results in thread runtimes.

I also tried to implement the advice on running the threads once and then leaving them running whilst I read in multiple queries however this just caused my threads to throw multiple errors, I believe that if the time is reset each time it should be an honest reflection on the actual runtime of the threads instead of keeping them running which may lead to inaccurate result times. As seen when printing the results it does not take a long time to run all queries which leads me to believe it is performing efficiently enough.

Magic Power

The runtime for execution of the third task (the magic power) on a large file is relatively slow and the time complexity is $O(n^2)$ as it has to go through each trace in the list and initially find all direct contacts of the magic power people. These people are added to a separate list and using two threads are split and all indirect contacts of these people are found.

We quantify the chances of getting the magic power as 10% for each direct contact you have with someone who is infected. Although it cannot be guaranteed that you will contract the power, after 10+ direct contacts we assume the chance is 95+%.

Formula: $P = 0.4D + 0.4^2I + 0.4^3I_2 + 0.4^4I_3 + \ldots$

- Where P is the probability of some person catching the power, D is the number of direct contacts they have had with someone who has the power, I is the number of indirect contacts they have had with someone who has the power, I_2 is the number of indirect, indirect contacts they have had with someone who has the power, etc.

We believe that a 40% chance of catching the power with direct contact is reasonable, based on how contagious COVID-19 is when in direct contact with it. This means that the probability of you catching the power from a single indirect contact is 0.4^2, or 16%, because there is a 40% chance of the direct contact getting the power, and then a 40% chance of you catching it from them, if they happened to catch it. The probability of catching the power from a single indirect, indirect contact is 0.4^3, or 6.4%, because there is a 16% chance of the indirect contact catching the power, and then a 40% chance of you catching it from them, if they happened to catch it, and so on.

In our code, we only implemented this formula up till the seventh indirect contact for small files, and up till the fourth indirect contact for larger files, because the probability of catching the power through this many indirect contacts becomes small, just 0.16% per contact for 6, and 2.5% for 3, and each extra level of indirect contacts reduces the run speed significantly. Because the large files go so deep and have so many contacts each we found a lot of the results ended up being over 100%, therefore we declared those individuals had a 95% chance of catching the power.