Jacob Cone, Castipher McSkimming | 3977920, 8287490

**Multithreading**

When multithreading, we start the threads by feeding in the list of people and times to a constructor contained within each thread (RunnableImpl), then call thread.start(), and immediately after call thread.join() within a try-catch loop. If we do not do this, we get an index out of bounds error as we start the next thread.
After this sequence of actions is completed, all the arraylists that have been created as a result of creating new threads are joined into one arraylist, allContacts.
The time begins just before we call thread.start() on the first thread, and the time ends just after we have joined all of the arraylists together.

Our program, in order to utilize more threads, halves or quarters the data we need to search so that it can be run concurrently with two or four threads instead of just one. Each of these lists of time and people is initialized before we start asking for commands.

To run our current code, there are three options:

java tracing filename [COMMAND] [PID] [time] [threadNumber]

ThreadNumber can be either 1, 2, or 4, as those are the only values we were testing with, and those were the recommended values in the Town Hall.

Command can be either pt, gt, mt, or st. For single people, you want to enter pt. If you enter gt, you enter as many PID as you want in the PID section. If you entered MT, enter [PID] [Time] over and over again, and do not enter a thread number, as magic tracing is set up only to run with single-threading, because the etude specifications only say to multithread for the first problem. Finally, to quit the program, just type "st" and the program will safely exit and finish.

No problem. Have added the below to the report as well.

To run gt tests, you need to type in gt, followed by all the PID you want to identify, then the time, then the multithread. So if you had three people, 4, 8 and 7, and you wanted to check starting from time 32 with 2 threads, you'd type:

gt 4 8 7 32 2

As for mt, you need to type in mt, followed by a PID, which is immediately followed by a time. Do this as many times as you'd like, and then finish with the number of threads. So if you had four magic people, 3, 2, 17 and 19, and they were activated at 7, 50, 23, and 100, you'd type:

mt 3 7 2 50 17 23 19 100

Jacob Cone, Castipher McSkimming | 3977920, 8287490

Below are all the times recorded.

We can look at the performance improvements from our system of processing here. For an example, we were mainly using the medium_trace.txt supplied, which has 9881 lines to process, which was split into halves (4940 / 4941) or quarters (2470 / 2470 / 2470 / 2471) for our multithreading tests. I performed five consecutive attempts on each test, and listed the minimum and maximum time from that five-test sample. While there might appear to be some overlap with minimum and maximum times, two threads were almost entirely consistently faster than single threading in the individual tests. The times below are in nanoseconds.

| **Test** | **Single-Threading** | **Two threads** | **Four threads** |
|---|---|---|---|
| pt 4 33 | Min: 4517300<br>Max: 5160600 | Min: 1570000<br>Max: 2832800 | Min: 1437500<br>Max: 2085200 |
| pt 8 54 | Min: 2646900<br>Max: 2927000 | Min: 1446000<br>Max: 1718000 | Min: 1352900<br>Max: 1804800 |
| gt 4 8 77 40 100 34 75 90 32 12 | Min: 5011000<br>Max: 74270500 | Min: 1343800<br>Max: 5369600 | Min: 1354900<br>Max: 4364000 |
| gt 12 12 3 4 7 90 | Min: 3038800<br>Max: 66144300 | Min: 1417400<br>Max: 1920600 | Min: 1396100<br>Max: 1497400 |

As you can see, the general trend is that the more threads I add, the quicker the program becomes at searching. Occasionally there is some overlap, but this is due to the CPU being variably weighed down by background processes - when testing times, you have to account for other things going on in the background. Additionally, there is extra overhead when you add more threads, which is why the first compile/run is occasionally slower, as it has extra overhead to do the initial setup of threads - so without the extra threads, two threads could sometimes pull a faster result just by virtue of the fact it doesn't have to start and join additional threads. But with large file sizes, single threads always seem to be slower for our searches. I didn't want to add any more than four threads, since my computer that I'm testing it on has only four physical cores. Any more than that would create an unnecessary burden on the speed of the program. If I create more threads than there are cores, we have five threads trying to fit through a space where only four can fit.

At one point we had a problem centered around a bottleneck, which was our data structure's use of two arraylists containing the people and the time - but these were consistently used by all of the threads, which realistically meant that every thread was trying to access the same thing at the same time, slowing our program down. In order to fix this, we added in a value that each thread takes in, indicating which number thread they were. Using that, we could just simply have a few if statements that allowed each specific thread access to their requisite arraylists, which meant there was no bottleneck, and they could all use them as required, fixing our previous

multithreading issue.

At one point we were having a problem with the times recorded by single-threading through a process that is known as super-linear speedup, which is where a problem is basically more effective split up than as one whole, where all the pieces were being executed more efficiently because of the fact they were in pieces. This was an odd occurrence, especially because single-threading was more than 2 seconds slower than multithreading. To fix this, we went into the code and examined the way in which we were single-threading, and cleared away a few extraneous lines of code that were slowing the single-threading down in a way the multithreading was not. We were using a few different if statements in our testing code to clear things away, which simply weren't necessary, as the requisite arraylists naturally reset by themselves or did not need to be reset at all. Finally, we were getting the results we needed for our multithreading, having done some clean-up to the code.

**Magic Power Likelihood**

To calculate the likelihood someone has met someone else with a magic power, we also needed to determine how many times someone had met someone else with a magic power, as we need to have a list ready to assign the counts to.

In order to calculate the likelihood of a magic power being present in an individual, we decided to go with a several HashMaps which keep track of people who have been contacted, but essentially sorting them into degrees of contact - the immediate targets are of course the most contagious, followed by a contact of a contact, then finally a contact of a contact of a contact. The value that is added to the corresponding chance is 1 if someone meets an immediate contact. If someone meets someone that was one degree removed (who has already met one of the immediate targets), we add 0.5 chance. If someone meets someone else who has been several steps removed, we add 0.25 chance. Our program implements these steps to calculate a person's likelihood of having a magic power.

Basically we want to have a base "power contagious chance" of 1 for anyone with a power at the beginning of our test. But then for the next person they meet, that chance is halved. And for the person that *that* person meets, the chance is again halved, and the process continues. This is because of the probability theorem, so you can see that we'll end up with an equation that looks like this:

**newChance = oldChance * 0.5**

,where newChance is the chance that will be assigned to the new person, and oldChance is the chance belonging to the person who has a magic power.

The further away we get from an individual who is actually powered, the lower the chance of them being able to pass on the power immediately is. So as the distance to an individual increases, the chance is lowered. That newChance value, once calculated, is added on to any subsequent meeting with another individual, whether it be 0.5, 0.25, or 0.125, etc.

**Take this example as an explanation.**

We have two magic individuals, 22, who was "activated" at time 12, and 50, who was activated at time 19. Then say we have person 4, who met person 22 at, say, time 30.

Person 4's chance is increased by 0.5. How is this possible? Look at our equation.

newChance = oldChance * 0.5

So our oldChance (the chance belonging to person 22) is 1. Multiplying that by 0.5 gives us our newChance for person 4 of 0.5.

newChance = 1 * 0.5 = 0.5

Person 4 is now sitting at 0.5 chance total currently.

Person 4 then goes on to meet person 45.

Person 45 meets person 4, but person 4 already has half the chance of one of the people with confirmed magic powers. Now we examine person 45's equation, with oldChance now being 0.5

newChance = 0.5 * 0.5 = 0.25

So person 45's chance is now 0.25, and that is added to their totalChance, which will be printed out at the end.

Person 45 goes on to contact person 97 and person 102. Person 97 has had contact with person 50 already, and so has a current value of 0.5 attached to them. However, person 45 only has a 0.25 chance, and they add a value of 0.25 onto person 97, giving person 97 a total of 0.75 chance. Person 102, who hasn't met anyone before, has a newChance of 0.125.

Basically in a chain of contact chance is multiplied, but if you add in a second chain, the two chains are added together to get the probability.

A more complicated problem would be that if we didn't have a time value, we could potentially have a long and complex loop of "who is affecting who", where individuals keep meeting and affecting each others' chance of magic powers. Fortunately for our more simple problem, we have time values to stop us worrying about that, and the time values are a way to check whether we need to add or not.

The process continues until we have gone through everyone and checked all the possible results. Our process could be said to undergo a certain amount of approximation, due to the fact we are keeping track of four layers of contact (the initial magic power users, the first degree of separation (0.5), the second degree of separation (0.25) and the third degree of separation

(0.125)). It is important to keep in mind that we do not need to go any further for this particular example due to the numbers being too small beyond 0.125 to really change particular chances, but we could if we needed to.

At the end, we return our HashMap of <String, Double> (name, chance), which can then either be combed through to search for specific instances if you had a need for that and had access to the source code, or currently can be processed and sorted by an EntrySet() for loop, which finds every HashMap coordinate, and can print out the results of each person and the score corresponding to their contacts. The higher the score, the more likely it is that they are contagious. Higher numbers indicate that a person has spent some time in the presence of a magical person, and is therefore likely to have that power.

Below you can see a short extract of sample output using the following command:
mt 45 67 8 12 3 44, where 45 was activated at 67, 8 was activated at 12, and 3 was activated at 44.

Person: 80
Likelihood of person having powers 12.375
Person: 81
Likelihood of person having powers 39.75
Person: 82
Likelihood of person having powers 19.5
Person: 83
Likelihood of person having powers 1.625
Person: 84
Likelihood of person having powers 0.625
Person: 85
Likelihood of person having powers 5.375
Person: 86
Likelihood of person having powers 4.375
Person: 87
Likelihood of person having powers 4.5

As you can see, out of this data set, person 84 is the least likely to have magic powers, whereas person 81 is the likeliest, as they had many encounters with magic people throughout their day, and are likely to develop powers later on.