SocialSim: An investigation on how information spreads in a social network

Max Barker

Abstract

The spread of information on a simulated social network has been investigated through the use of differing data structures and algorithms in the hopes of uncovering the way in which information spreads in a real life social networks. This investigation has been conducted to explore what time complexities different algorithms use, and in what manner information spreads, be it linearly or exponentially. We will also explore how the spread of information will affect the memory needed by the simulation machine.

To do this, a Java program called SocialSim has been created, in which a network of users and their followers have the opportunity to like posts and follow each other based on a given probability. Time is recorded in an algorithm called a time step, which allows posts to flow outwards into the network and spread to its edges. Through this simulation, we have observed the manner in which this information spreads.

It was found that SocialSims execution time would grow exponentially (O(N^2)) depending on the probability of a post being liked. The individual timesteps however would increase linearly (O(N)) as the information flowed further through the network. It was also found that the memory needed for a 1.0 probability simulation was dramatically higher than one with only a 0.5 probability (5MB vs 100MB of RAM).

The findings show how rapidly information can spread throughout a social network. This can be seen in real life examples, where the virality of a post on a social network can influence how many people are exposed to it. Very quickly, a post could go from person reached to millions due to the exponential nature of the growth. From here, the simulations will continue to be tested with larger data pools and observational data should be collected from posts within real life social networks.

Background

The approach that SocialSim took towards creating a simulation mode took a lot of planning. The simulation required an effective time step algorithm in order to work. A time step in SocialSim can be described as an outward flow from each individual post. When a users post flows outwards into the network, the users followers have a chance to like it. If those followers like it, the post flows out into the scope of their followers, in which the post has a probability of being liked again, and the original user being followed. Then, if we call another timestep, we can flow outward even further from the original post. The post also stores who decided to like the post, so that they cannot like it again on repeated timestep calls.

The timestep is a large algorithm that relies heavily on iterators. Put simply, during the timestep, the simulator iterates through the post list. If a post is found, then we get the user that created the post (retrieved from a getUser() method) and search for said user in the Vertice list. If they are found, then we iterate through each users adjacent users (followers) and calculate the probability of their followers liking it through a probability() method. In this method, we take the Double probability given in the command line and use the Math.random() built-in java function. If number returned from Math.random is less than the probability, then the follower is said to have liked the post. The number of likes is incremented in the post, and the user's name is stored in a LinkedList of String names representing the users. Another iteration occurs, this time through the followers of the users followers (mutual friends). In much the same process as before, we run a probability that the users mutual friends also like their post. If they do, then we add another like to the Post, and then run the probability that they follow the user. If the mutual friends fall under the probability, then we create an edge between the user and the new poster.

The next time a timestep is called, the original poster may have acquired new followers. If this is the case, then those followers, through the same algorithm, have a chance to expose it to their followers, and so on for each timestep. Two of the conditions of a timestep forbid a user from liking their own post and for a person to like a post twice.

When the simulation is initially run, the network file is loaded into the graph network. The simulator also takes note of the current time the program has been running for in milliseconds, and stores it. The graph network contains LinkedLists of Vertices and Edges, each having their own classes. The vertice list is populated with vertices that represent users, and the Edge list is populated with edges that represent users who follow each other. Once successfully loaded, the simulator loads the event file into the graph network. Each line of the event file is parsed and stored into an array. The first index of the array tells the simulator what to do. If the line contains a ‘P’ then a new post will be created based off the rest of the line. The created Post object is then added into a LinkedList stored in the Network that stores post objects. After all of this, a time step is run. After the timestep, the current state of the network is written to a file. This process is repeated until the file reaches its end. Once we reach the end, the execution time is taken again, with the start time subtracted from it. The simulation time in milliseconds is then printed to the terminal.

Methodology

SocialSim uses a number of algorithms which will affect its overall time complexity. For the most part, the algorithms within use O(N) time complexity due to the use of LinkedLists, however, SelectionSort (O(N^2)) has been implemented to find the post with the most likes. The effect of having multiple O(N) algorithms will have on the simulations overall time complexity is presently unknown. Due to the large amount of searching through LinkedLists that will be occurring during the simulation, the memory use of the machine will be recorded.

The methodology aims to find out what kind of time complexity we are dealing with in the simulation, and how it affects the machines memory. I hypothesise that as the probability increases, the execution time will grow exponentially. I also predict that the execution time for each timestep is O(N^2) time complexity.

To analyse the time complexity, the provided Toy Story network and event file will be used. Even probabilities (0.2 , 0.4, … 1.0) for both the like and follow probabilities will be supplied in the command line and the execution time will be noted and taken down for analysing. A sample size of 20 will be used for each different probability. The average times will be taken and plotted graphically which will reveal a suspected O(N^2) time complexity.

The timestep algorithm uses multiple iterators that will iterate through LinkedLists. This means that we have an O(N) worse case scenario. In some instances, the entire list needs to be traversed, such as going through every single post to update the likes, or every single vertex's adjacent vertex. This means that as the network grows larger through the simulation, the time it takes to perform a timestep will be increased.

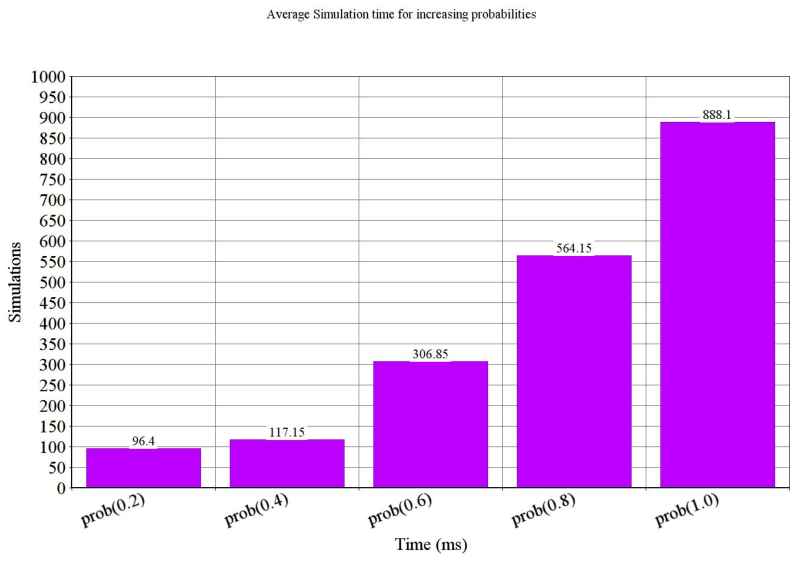
The command format run the simulation is as follows: `java SocialInterface -s network_file.txt event_file.txt prob_like prob_follow`. The command that was used to run the tests with the experiment specific network file and event file is as follows: `java SocialInterface -s ToyNetwork.txt ToyEvent.txt 0.2 0.2` . Note that the prob_like and prob_follow arguments were incremented by .2 each test. Once you run this command, the simulation states are saved to a folder called Files and the execution time will be printed on the terminal for both the individual timestep execution time and the overall execution time.

Memory use will be checked during the process using htop, a command line display that shows the memory use at a given moment. To test the memory, the initial memory use of the machine will be taken down. Once the simulation is running, the memory use will be recorded and compared to the memory before hand. This will be done for both a 0.5 and 1.0 probability. The memory will most likely spike once the simulation is running, reaching its peak towards the end of the file. It will probably use little to no memory in total, with a suspected 10mb for 0.5 probability and 20mb for 1.0 probability.

To ensure a controlled environment, the machine used to run the simulator will have all other processes killed to ensure that both the memory use and execution time is not compromised by external factors. All wireless connections will be disconnected. Any outside influences will destroy the integrity of the testing.

Results

The testing found that SocialSim exponentially takes longer the larger the data pool (O(N ^2)). The first iterations with a 0.2 probability had quite predictable results (around 100 ms) with the occasional drop (40ms). Fluctuations like this can be found up until the 1.0 probability multiplier. This would be due to the fact that due to the fact that we're working with probabilities sometimes the execution is faster than previous runs.

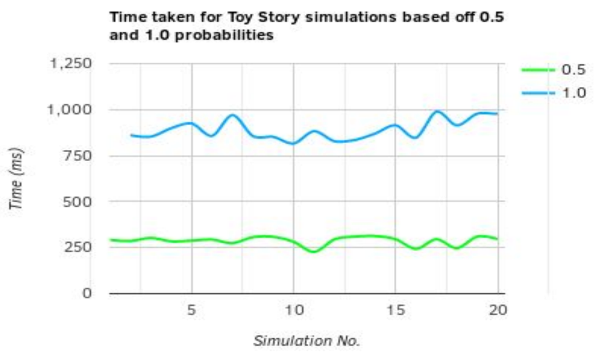


As seen in figure 1.0, the millisecond time to execution increases exponentially as the probability is higher. This O(N^2) time complexity is obviously due to the fact that each timestep flows further outward further and further into the network This is the outward flow that was described earlier. When the simulation is reaching its end, there are so many LinkedLists that are being iterated that over that the initial O(N) time complexity of a LinkedList factored ontop of eachother makes the program take exponential time to get done.

Unfortunately, testing this exponentiality cannot continue past the point of 1.0, as that is the maximum probability you could have in a real world setting. However, perhaps with a larger data files we could see a different time complexity.

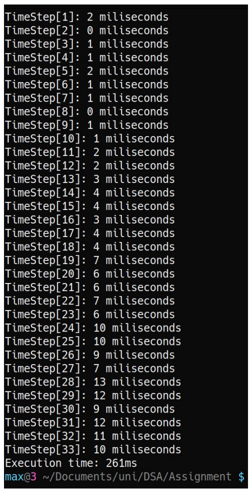
[Fig 1.0]

To give an example of the variability of the results, 20 tests were run for both 0.5 and 1.0 probability levels. Each individual execution time was plotted onto a graph (see figure 1.1). Both probabilities show a similar variability, with the 1.0 being slightly more variable. This variability in 0.5 probability can be explained by the fact that there is an element of luck involved. We may get more likes per timestep on one simulation than we do on another. However, with a probability of 1.0, the variability is a bit more of a mystery. There should be a 100% chance of likers and followers, whi.ch would suggest a very similar time for each simulation, but there isn't. Perhaps this can be explained by the fact that more CPU cores and RAM are needed to execute the program.



[Fig 1.1]

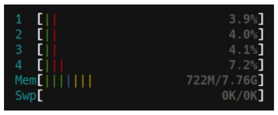
[Fig 1.2]



```
TimeStep[1]: 2 miliseconds
TimeStep[2]: 0 miliseconds
TimeStep[3]: 1 miliseconds
TimeStep[4]: 1 miliseconds
TimeStep[5]: 2 miliseconds
TimeStep[6]: 1 miliseconds
TimeStep[7]: 1 miliseconds
TimeStep[8]: 0 miliseconds
TimeStep[9]: 1 miliseconds
TimeStep[10]: 1 miliseconds
TimeStep[11]: 2 miliseconds
TimeStep[12]: 2 miliseconds
TimeStep[13]: 3 miliseconds
TimeStep[14]: 4 miliseconds
TimeStep[15]: 4 miliseconds
TimeStep[16]: 3 miliseconds
TimeStep[17]: 4 miliseconds
TimeStep[18]: 4 miliseconds
TimeStep[19]: 7 miliseconds
TimeStep[20]: 6 miliseconds
TimeStep[21]: 6 miliseconds
TimeStep[22]: 7 miliseconds
TimeStep[23]: 6 miliseconds
TimeStep[24]: 10 miliseconds
TimeStep[25]: 10 miliseconds
TimeStep[26]: 9 miliseconds
TimeStep[27]: 7 miliseconds
TimeStep[28]: 13 miliseconds
TimeStep[29]: 12 miliseconds
TimeStep[30]: 9 miliseconds
TimeStep[31]: 12 miliseconds
TimeStep[32]: 11 miliseconds
TimeStep[33]: 10 miliseconds
Execution time: 261ms
max@3 ~/Documents/uni/DSA/Assignment $
```

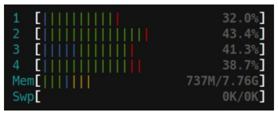
In figure 1.2 the terminal output of a simulation is shown. Each output is representative of a new Post being added to the network, a timestep being called, and a file being written to represent each timestep. The time for each individual timestep is also printed. The times vary yet increase as the timeseps are called more. The milliseconds taken in this figure are quite linear, as opposed to the exponential time taken when comparing simulations. It becomes clear from this that the individual simulations are working on O(N) time complexity individually, as in the process of running a simulation takes linear time. However, when we take all of these individual process' with different probabilities we can see an O(N^2) time complexity. Initially it was expected that both the individual simulation and the comparisons between them would be O(N^2), but the data has revealed otherwise.

The effect on the machines memory was also quite noticeable for such a small process. An additional piece of data we got to measure was the effect the simulations had on the machines core use, of there being 4.

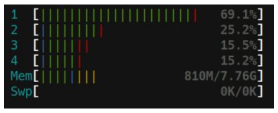


```
1  [|                 3.9%]
2  [|                 4.0%]
3  [|                 4.1%]
4  [||                7.2%]
Mem[|||||||     722M/7.76G]
Swp[                 0K/0K]
```

In the figures 2.0, we can see the states that machine was in before during and at the end of the simulation running a probability of 1.0. Almost 100mb of memory were needed out of the machines RAM which is quite significant for such a small program. The core use also went up evenly initially, but when the simulation reached its peak core 1 neared its max as cores 2 - 4 dropped slightly. The reason for this behaviour between the cores is unknown.



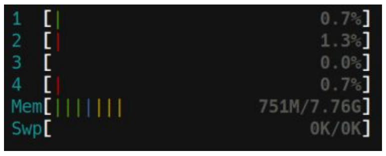
```
1  [||||||||||       32.0%]
2  [||||||||||||||   43.4%]
3  [|||||||||||||    41.3%]
4  [|||||||||||||    38.7%]
Mem[|||||||     737M/7.76G]
Swp[                 0K/0K]
```



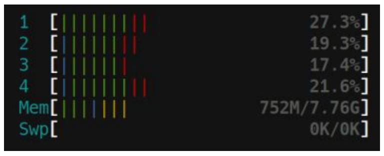
```
1  [|||||||||||||||||||||  69.1%]
2  [||||||||              25.2%]
3  [||||||                15.5%]
4  [||||||                15.2%]
Mem[|||||||||       810M/7.76G]
Swp[                     0K/0K]
```

From this, we can see that a large scale social network with potentially billions of users would require gigantic levels of processing power. However, this simulation is not a perfect simulation. We are investigating the spread of information, however, the fact that we are doing this immediately is unlike real life a Social network runs in real time.

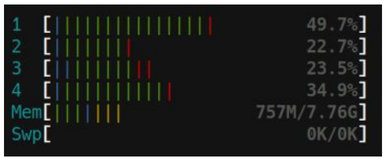
In figure 2.1, we can see the memory use of a 0.5 probability simulation. This memory use is much more in-line with what was to be expected, as compared to figure 2.0. The memory increase is much, much lower than a probability 1 simulation. Only 5mb were used, compared to 100mb. This is a huge leap, and goes to show that the O(N^2) time complexity of the simulation is affecting the memory demand significantly.



```
1  [|                 0.7%]
2  [|                 1.3%]
3  [                  0.0%]
4  [|                 0.7%]
Mem[|||||||     751M/7.76G]
Swp[                 0K/0K]
```



```
1  [||||||||         27.3%]
2  [|||||||          19.3%]
3  [||||||           17.4%]
4  [||||||||         21.6%]
Mem[|||||||     752M/7.76G]
Swp[                 0K/0K]
```



```
1  [||||||||||||||||   49.7%]
2  [|||||||            22.7%]
3  [|||||||||          23.5%]
4  [||||||||||||       34.9%]
Mem[|||||||       757M/7.76G]
Swp[                   0K/0K]
```

Conclusion

Initially, I predicted that when comparing different simulations with their own probabilities, we would find an exponential increase in their execution time. What I did not expect was the linear execution time increase for each timestep within each simulation. A potential reason for this would be the fact that, while it does take longer for the lists to be iterated through when liking posts and adding edges, the time doesn't increase exponentially due to the fact that more connections are made, therefore, less iterating needed and higher chances that a post has already been liked

The effect I predicted the simulator would have on the machines memory really surprised me. Rather than an expected 20MB increase in RAM for the 1.0 probability, I instead found a 100MB increase. This is much higher than the 0.5's 5MB increase! The memory used doesn't surprise me, rather it's a massive difference between the two that surprises me.

Further investigations that follow should be the effect that a much larger file would have on the simulator, perhaps one with upwards of a thousand users. Taking this investigation further than that may require a complete remodel of our approach, as the current simulator that has been created doesn't really show us anything in a real life scenario other than the connections that could be made.