MP1: Performance Study

Authors: Steve Huang, Asher Kang, Maria Ringes

I. Introduction

This program is our implementation of MP1, which demands a program that simulates and analyzes the spread of three types of Gossip through a synchronous network of nodes, using Go-routines and Go-channels. The three types of Gossip simulated are Push-based Gossip, Pull-based Gossip, and Push/Pull-based Gossip.

This program asks for three inputs from the user: the type of Gossip to be simulated, the maximum size of the network, and whether to show the verbose output. The program then iterates from 1 to the desired maximum size, incrementing by 1 each time. In each iteration, the program initiates a network of the given size and simulates the wanted type of Gossip. The program finishes when the spreading of Gossip has been completed on the largest size of network. The output consists of a graph displaying the Gossip's Convergence Time vs. the amount of nodes. If the user chooses to print the verbose output, the status of each network at each round will also be displayed.

In addition to the three required types of Gossip, this program also implements a fourth type of Gossip: Push/Pull Switch. This protocol begins by executing the Push protocol and switches to the Pull protocol as soon as the Gossip has spread across half of the network.

II. Methodology

A. Research

Our first step in research was understanding the Gossip protocol. Gossip refers to the spread of information via peer-to-peer interaction in a network of people. Every person in the network has either heard the Gossip and is actively spreading it, or has not received it and is susceptible to receiving it. The spread of Gossip is analogous to the spread of a virus, in that the rumor is a "virus" that is transmitted from node to node, resulting in the "infection" of the entire system. Gossip is of interest to the field of distributed systems because it demonstrates the distributed, decentralized spread of information in a system of nodes. Understanding Gossip and its protocol types helps us to understand how nodes can pass and process information to each other in a decentralized, efficient way, and update the entire network.

There are three major types of Gossip protocols: Push-based Gossip, Pull-based Gossip, and Push/Pull-based Gossip. These protocols take different approaches to node-to-node interaction and passing of infection. In Push-based Gossip, each infected node randomly selects a peer node in the system and passes the Gossip to it. In Pull-based Gossip, each susceptible node randomly selects a peer node and executes a "pull" from the peer, thereby becoming infected if the peer happens to be infected. In Push/Pull-based Gossip, both Push and Pull protocols are used in each

infection round: each node randomly selects a peer node and executes both a “push” of the infection to the peer if it is infected, and a “pull” from the peer.

We based our Gossip algorithms off of the General Algorithm presented by Márk Jelasity (Jelasity, 6). In this General Algorithm, each node runs through the loop procedure and randomly selects a peer node. Depending on the Gossip protocol being run, the node either calls the Push or Pull procedure.

Algorithm 1 SI gossip

```
1:  loop
2:      wait(Δ)
3:      p ← random peer
4:      if push and in state I then
5:          send update to p
6:      end if
7:      if pull then
8:          send update-request to p
9:      end if
10: end loop

11: procedure ONUPDATE(m)
12:     store m.update      ▷ means switching to state I
13: end procedure
14:
15: procedure ONUPDATEREQUEST(m)
16:     if in state I then
17:         send update to m.sender
18:     end if
19: end procedure
```

Gossip General Algorithm (Jelasity, 6)

Our second step in research was understanding the convergence times of all three types of Gossip. Convergence time is measured as the number of infection rounds it takes to infect the entire system of nodes. As outlined by Jelasity (7-8), the convergence time for all three types of Gossip is O(log N). However, we expected to see a difference in convergence times in practice because random peer node selection, which determines the advancement of Gossip across the system, is heavily influenced by the current phase of the Gossip spread. Gossip spread has two major phases: either the number of susceptible nodes is greater than the number of infected nodes, and or the number of infected nodes is greater than the number of susceptible nodes. In the first phase, Push-based Gossip is optimal, as infected nodes are more likely to select and infect susceptible nodes than already-infected nodes. In the second phase, Pull-based Gossip is optimal, as susceptible nodes are more likely to select and receive the infection from infected nodes than other susceptible nodes.

B. Data Collection

Printed Details

For each round, if the user desires, as indicated by the 3rd question asked of the user, details are printed using simple print statements. These print statements output the following information: when a new round begins, when a new phase begins, which node is being infected (as decided by the random peer node selection), when a node has completed an infection, when the channels are cleared, when the completion check occurs, and finally, the infection status of each node (as

received by looping through the **nodes[]** splice of Nodes). Finally, after each network of nodes has been fully infected, the program prints how many rounds it took to finish the set number of nodes for that network.

Output Plot

Each time our program is run, data necessary for an analysis is output, rather than just data for one system of nodes. For example, if the user inputs a desired 50 nodes, the program collects and outputs data for a 1-node system, then a 2-node system, 3-node system, and so on, culminating in a 50-node system.

To collect, store, and output this data, an empty global splice named **desiredNodesResults** is initialized. The program uses this splice like a map, but instead of key and value pairs, it uses the indices and the values of those indices. Our program first appends 0 to this splice because a system with 0 nodes (0 index) takes 0 rounds to terminate.

Once the user input has been collected and parsed into variables, our program calls **initiateGossip()**. This function executes a for-loop that runs for each network of nodes a user would like to test (1,2,3,4...n-node system). For each network of nodes, our program initializes all the nodes and associated channels. The variable **roundCount** is initialized and set to 0. The Gossip protocol rounds then begin. In this for-loop of Gossip protocol rounds, the program first adds 1 to the roundCount. Then, the Gossip protocol type (as determined by the user) is called. After the Gossip protocol has been completed, the program checks if all the nodes have been infected. If they have all been infected, no more rounds are needed. Therefore, the program breaks from the inner for-loop that continuously calls the Gossip protocol rounds. At the end of the larger for-loop that runs through each system of nodes, the current **roundCount** is appended to the **desiredNodesResults** splice. After the **roundCount** for the user-input *n-node* system has been appended, the larger for-loop breaks.

The **Plot()** function is then called, and uses the **desiredNodesResults** splice to plot the **roundCount** values with their corresponding indexed number of nodes. For example, desiredNodesResults[5] is the number of rounds it took for a system with 5 nodes to become fully infected. The **go-echarts** package is used to materialize the plot of this data. The program assigns keys and values to be plotted as scatter points. This first for-loop in this plot function fills a splice named **keys** with all the completed indices of the **desiredNodesResults** (0...n, as the user indicated). It also fills a splice of **opts.ScatterData** named **values** with the data from **desiredNodesResults**. The keys and values splices are plotted as points, with the **key** value as the x-value and the **values** value as the y-value.

The system then checks whether the current directory is writable. If this check is successful, an HTML file is created with the rendered scatter plot.

Time Efficiency Analysis

Our motivation in implementing the Push/Pull Switch was to find a method that would be more time efficient than that of Push/Pull. To prove this, we performed an analysis on program runtime versus Gossip protocol. The program was run 40 different times. Each run tested networks through 500 nodes. Additionally, no round details were printed. 10 of these runs ran the Push protocol, 10 ran the Pull protocol, 10 ran the Push/Pull protocol, and 10 ran the Push/Pull Switch protocol. The complete program runtime, as well as the total number of rounds, was output and stored in a table, attached in the Appendix section. From this, the time per round was also calculated. The averages for each Gossip protocol were then calculated.

C. Tools Used

The "e-charts" package allowed us to plot the values defining the comparison between number of nodes in a network and the number of rounds it took to infect all those nodes, starting with one infected.

D. Design Decisions

First, we chose to design the system's simplest component, a node. Each node has an **infected** boolean as well as a channel pointer, which points to the node's associated channel. Pointers were used to allow multiple locations to access these channels. A node's associated channel would be initialized within the **initiateGossip()** function. We decided to use just one channel, for at one time, only one Gossip function would be running (**pushInfect()**, **pushUpdate()**, **pullInfect()**, **pullUpdate()**). Channels used in this way could be analogous to the mailbox of a house, wherein mail can be sent from and sent to.

To simulate a synchronous system, we chose to run each Gossip protocol in lock step. This lock step was implemented using the **sync.WaitGroups** package. For example, in the push Gossip, a wait group would wait for all push infections to occur before any push updates would occur. This lockstep would prevent the program from missing any infections. If these functions had not been synced, a node may have updated its stage before it had been infected by another node, which then would have caused our program to have missed reading that infection. Further, clearing channels was another function used in the lockstep process. This was again to prevent our system from missing an infection. If a node's channel had been cleared before the node was able to update its infected state, the infection would be missed. For this reason, it was important to make sure each function occurred in lockstep. We also considered utilizing a buffered channel to execute the synchronization, however this design was not preferred. In our opinion, wait groups offered a clearer visualization of synchrony.

Another design choice was that of clearing channels. Originally, the values of a node's channel would be read and stored as no variable. This design choice seemed to be not useful, for each value was just being merely deleted. Rather, we chose to use these values to check our implementation. Each value of the channel would be logically OR-ed with each of the others.

Only if every value in the node's channel was **false**, would the node become uninfected. Because our system is designed to never send a **false** value, this design choice was not one of great necessity, however, if someone were to hack and change the design, this design would allow us to better debug the hacked solution.

The last design choice was to implement the fourth Gossip protocol referenced as Pull/Pull Switch. Gossip, in its most standard form, has 3 different protocols: Push, Pull and Push/Pull. In these protocols, there is no consideration of the number of susceptible nodes. For this reason, we chose to implement a fourth Gossip protocol that would be more time efficient while staying efficient in its Gossip implementation. In the push protocol, the amount of rounds it takes to infect the last node could be large for there is only 1/n chance that the n-1 nodes select the last susceptible node. In the pull protocol, the amount of rounds it takes to infect the second node could be large for there is only a 1/n chance that a susceptible node would pull from an infected node. In the Push/Pull protocol, push and pull are both run in each round. Although this method increases the probability of infecting a susceptible node in one round, this method will require more time per round. Therefore, we choose to implement a Push/Pull Switch protocol. This protocol would first begin by calling just the push Gossip in each round. Once at least half the nodes in the network are infected, just the pull Gossip protocol will be run in each round. This design was chosen due to the probability of infecting a susceptible node in each round.

If the number of susceptible nodes > number of infected nodes, the probability of selecting a susceptible node is greater than the probability of selecting an infected node. Because the push Gossip protocol works by randomly targeting a susceptible node, the push Gossip protocol will have a higher chance of infecting a susceptible node in one round.

If the number of susceptible nodes < number of infected nodes, the probability of selecting a susceptible node is less than the probability of selecting an infected node. Because the pull Gossip protocol works by randomly targeting an infected node, the pull Gossip protocol will have a higher chance of infecting a susceptible node in one round.

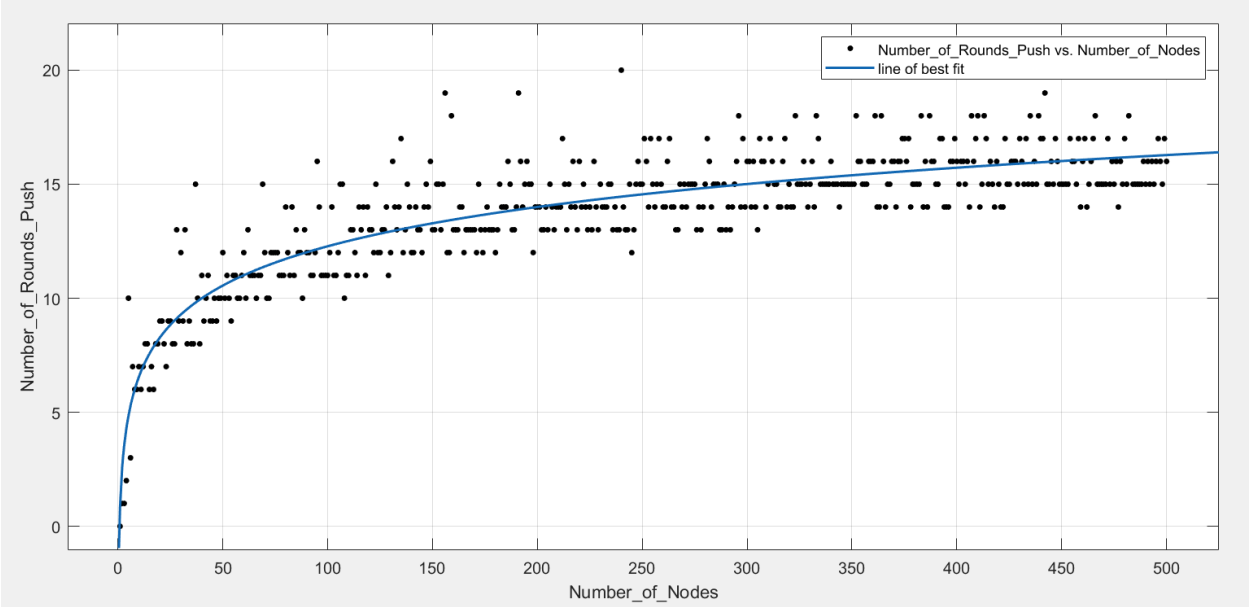
If the number of susceptible nodes is equal to the number of infected nodes, the probability of selecting a susceptible node is equal to the probability of selecting an infected node. In our design, we chose, in this scenario, to implement the pull protocol, however the push protocol would also have been a valid choice.

III. Results

Gossip Type Convergence Time Analysis

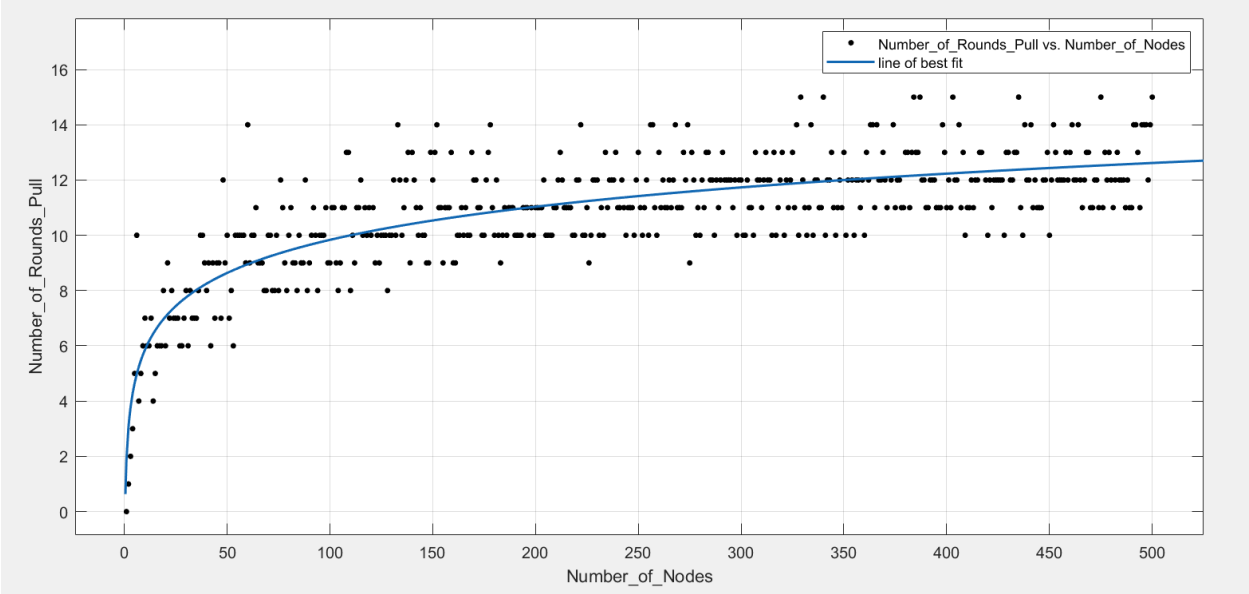
Choosing a relatively large number of nodes (n=500), we observed that the growth of convergence time (number of rounds) vs. the size of the network appeared to be logarithmic for all different Gossip types. In other words, the convergence time of all Gossip type algorithms is $O(logN)$. To compare the difference between Gossip types, we fit a logarithmic function $y(x) = a \times log(x) + b$ to the output graph of each Gossip type. The resulting graphs are as follows:

Push-based Gossip



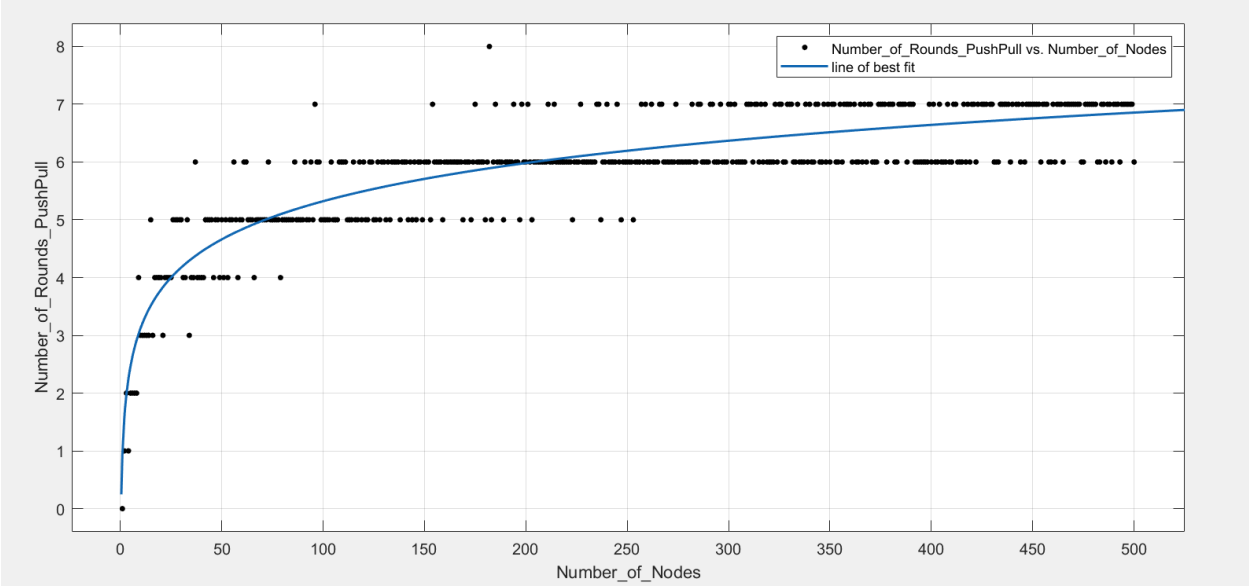
$$y(x) = 2.485log(x) + 0.8317,\ (r^2 = 0.7609)$$

Pull-based Gossip



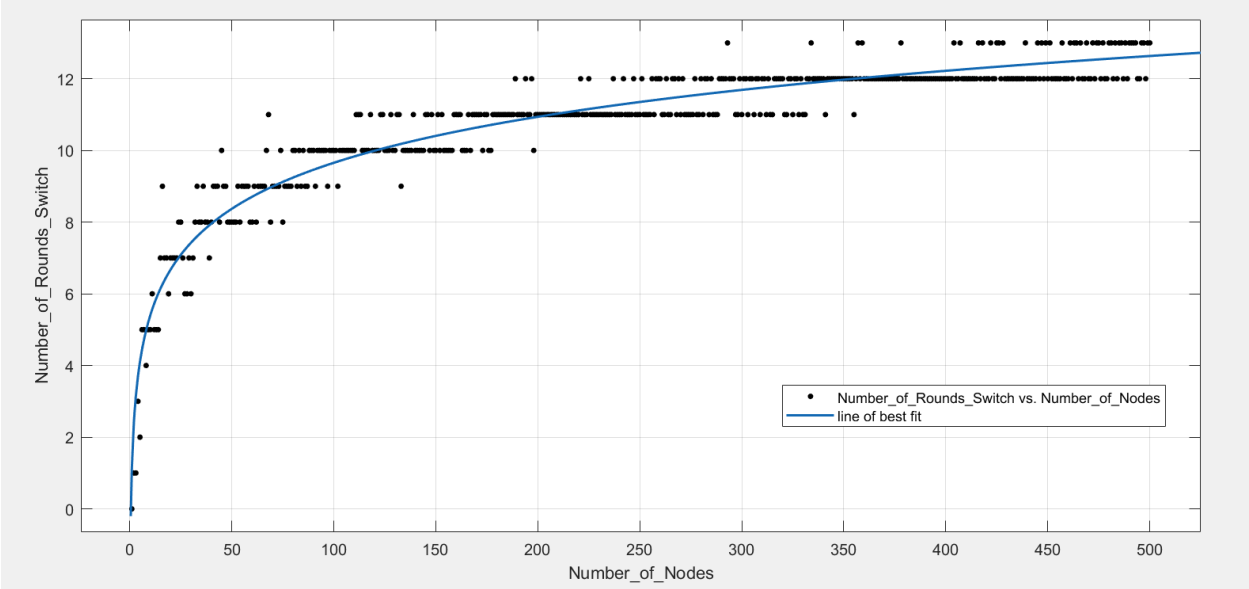
$$y(x) = 1.728log(x) + 1.876,\ (r^2 = 0.6281)$$

Push/Pull-based Gossip:



$$y(x) = 0.9531log(x) + 0.9322,\ (r^2 = 0.7536)$$

Push/Pull Switch:



$$y(x) = 1.851log(x) + 1.129,\ (r^2 = 0.9181)$$

Despite the relatively low r^2 value, these lines are still a good indication of the general trend of the data. Because of the spread of these data points, a higher r^2 value cannot be achieved without the risk of overfitting.

The a value of each function indicates the speed at which the convergence time will grow in relation to the size of the network. A high a value suggests that the Gossip type will take more rounds to complete given a network of fixed size.

Program Runtime Efficiency Analysis

To reduce the effect of the outliers on the output and properly examine the time taken by each Gossip type, we measured the program runtime (in seconds) of each type of Gossip and the total amount of rounds taken (excluding the time taken to process inputs). We then divide the measured runtime and divide it by the total number of rounds to obtain the average time (in milliseconds) taken for each round. The summary of results is as follows:

Averages	Push	Pull	Push/Pull	Push/Pull Switch
Program Run Time(s)	4.4258	3.585	5.3861	3.4683
Rounds	6800.6	5466.1	2966.7	5431.5
Time/Round (ms)	0.658081687	0.6558639681	1.815980464	0.6385721522

Summary of Time Efficiency of Gossip Protocol Types

Full analysis details can be seen in the Appendix section below.

IV. Discussion

We can conclude from these results that Push/Pull-based Gossip is the most effective protocol in terms of convergence time. This was expected because Push/Pull takes advantage of optimal random peer node selection during both the first and second phases of Gossip spread. This conclusion matches that of Jelasity, who notes that Push/Pull Gossip is faster than Push and Pull in practice (8).

In terms of program runtime, Push/Pull Switch is the most efficient protocol. Even though it takes on average 1.8x more rounds than Push/Pull, each round is run in about 1/3 the amount of time of that of Push/Pull.

V. References

Jelasity, Márk. “Gossip.” *Self-Organizing Software – From Natural to Artificial Adaptation*, Springer , Berlin Heidelberg, Germany, 2011, pp. 1–8.

VI. Appendix

Time Efficiency of Gossip Protocol Types

Gossip Method	Push			Pull			Push/Pull			Push/Pull Switch		
Number of Nodes	500			500			500			500		
Printed Results?	N			N			N			N		
	Time (s)	Rounds	Time Per Round (ms)	Time (s)	Rounds	Time Per Round (ms)	Time (s)	Rounds	Time Per Round (ms)	Time (s)	Rounds	Time Per Round (ms)
Run 1	5.445	5477	0.9941573854	3.418	5477	0.6240642688	5.314	2962	1.794058069	3.286	5429	0.6052680052
Run 2	3.483	6914	0.503760486	3.517	5469	0.6430791735	6.451	2959	2.180128422	3.928	5436	0.7225901398
Run 3	4.952	6890	0.7187227866	3.714	5483	0.6773664053	5.339	2974	1.795225286	2.801	5458	0.5131916453
Run 4	6.922	6941	0.9972626423	3.04	5464	0.5563689605	5.275	2964	1.779689609	3.745	5405	0.6928769658
Run 5	3.888	6934	0.5607153158	2.962	5523	0.536302734	5.404	2966	1.821982468	3.166	5423	0.5838096994
Run 6	3.652	6995	0.5220872051	3.071	5424	0.5661873156	4.775	2964	1.61099865	3.393	5412	0.626940133
Run 7	3.738	6934	0.5390827805	3.02	5434	0.5557600294	4.992	2994	1.667334669	3.838	5451	0.7040909925
Run 8	3.387	6934	0.4884626478	4.996	5471	0.913178578	6.331	2949	2.146829434	3.658	5455	0.6705774519
Run 9	3.556	6980	0.5094555874	3.51	5460	0.6428571429	4.42	2971	1.487714574	3.735	5427	0.688225539
Run 10	5.235	7007	0.7471100328	4.602	5456	0.8434750733	5.56	2964	1.875843455	3.133	5419	0.5781509504
Average	4.4258	6800.6	0.6580816 87	3.585	5466.1	0.6558639681	5.3861	2966.7	1.815980464	3.4683	5431.5	0.6385721522
Minimum	3.387	5477	0.4884626478	2.962	5424	0.536302734	4.42	2949	1.487714574	2.801	5405	0.5131916453
Maximum	6.922	7007	0.99726	4.996	5523	0.9131	6.451	2994	2.180128	3.928	5458	0.722590

			26423			78578			422			1398