FastLSU: A Linear Implementation for Controlling the False Discovery Rate

Mckervin Ceme

2016

Advised by Professor Sandra Batista

Submitted to: Princeton University

Department of Computer Science

This Thesis represents my own work in accordance with University

Regulations

Date of Submission: April 29, 2016

Abstract

Large-scale hypothesis testing is an important experimental procedure in the field of bioinformatics. Often times, researchers would like to control the rate of false positives that appear in their experiments. A current method to control this rate - false discovery rate - is known as Linear Step Up from Benjamini and Hochberg (1995). Linear Step up does control the false discovery rate, but it has a linearithmic time complexity, which does not scale well for tests on the order of 10^7 or larger. Professors Vered Madar and Sandra Batista proposed an alternative algorithm called FastLSU that applies linear scans over the data set to control the false discovery rate, albeit in a linear fashion as opposed to a linearithmic one. In this project, we aimed to successfully implement the algorithm for use on individual machines in a manner that maximizes efficiency while minimizing memory usage and created a simple graphical user interface that researchers can easily use to control the false discovery rate in their experiments.

1. Introduction

In the field of bioinformatics, researches often have to analyze large quantities of hypotheses in a given experiment. Generally, these type of analyses are called *large-scale hypotheses testing*, in which hundreds of thousands of different “hypotheses” (represented as *p-values*) are processed in order to determine which of the tests are significant in a particular experiment, such as when researchers perform Genome Wide Association Studies (GWAS) to determine particular traits or diseases among different populations of people. When performing these type of experiments, it is important for researchers to be able to determine the proportion, or rate, of "false positive" tests (or Type 1 errors, as they are called in statistics) in a given data set. This problem of determining the rate of Type 1 errors is called the *multiple comparisons problem*. In principle, when trying to control this rate, the tester would pick a significance level α in the range [0, 1] which determines the amount of false positives one might expect. Even with the selection of a proper α, there is still a need to try to control, or minimize, the amount of false positives in a data set. If for example, $\alpha = 0.05$, then one would expect approximately 5% Type 1 errors. As the size of the data set increases, this 5% represents significantly more hypotheses. Concretely, if there were 100 tests in a given experiment, then 5 of them would be tolerated as false positives. But if there were 100,000 tests, then 5,000 could be prone to error - this would not be an ideal situation. So to resolve this problem, statisticians have developed numerous methods in order to address the problem. While there have been other solutions proposed by other computer scientists, these methods all require a linearithmic sort of the dataset prior to determining the significant tests. Also, they do not return accurate results if the data is segmented.

The goal of this project is to use a newer algorithm called FastLSU – developed conjointly by Madar and Batista [7] – and implement the algorithm and ultimately create a simple tool that will allow researchers to control the rate of false positives while also doing so much more efficiently. FastLSU allows an input experiment to be segmented into an arbitrarily-sized segments and then controls the rate of false discoveries in linear time. In addition, the FastLSU algorithm does not affect the results from the original Linear Step Up: both the naïve method and this new procedure produce equivalent results.

1.1. Road Map

Throughout this paper, we intend to show how the FastLSU algorithm allows for significant performance gains to that of the original Linear Step Up procedure. We first define how controlling the rate of Type 1 errors had been done in the past, after which we later explain in detail how the FastLSU procedure is a) linear in run time complexity and b) able to operate under different kinds of constraints. Given this, we present our GUI application that performs the procedure efficiently. In order to justify our claims, we present memory profiling data as well as timing analyses for the final application as well as timing analyses that we used to justify some of the design decisions that we ultimately made when implementing the procedure. From all this, at the end of this paper (Section 10) we use the evidence presented to argue that our implementation achieves the linear time complexity both in space and in runtime while also being easy-to-use for non—developers. Towards the end, we propose some future improvements to the tool that would be able to improve the user-experience both in terms of runtime and in terms of aesthetics.

2. Previous Work

2.1. Benjamini-Hochberg

A few prior solutions to the *multiple comparisons problem* have been proposed. One of which is the Linear Step Up procedure, which controls *false discovery rate* (FDR) of a given data set. Linear Step Up, first proposed by Benjamini and Hochberg [5] in 1995, is a procedure that was designed to control the proportion of Type 1 errors – "false positives" – in the set of rejected hypotheses (i.e. control the false discovery rate for a given experiment). The following table is helpful when trying to mathematically define the false discovery rate (here, H_0 is the null hypothesis, and m is the number of tests):

	Accept H_0	Reject H_0	Total tests
H_0 true	U	V	m_0
H_0 false	T	S	m_1
...	...	...	m_i
	W	R	m_{n-1}

Table 1: Outcomes from multiple hypothesis tests

Using this table, we can define the FDR mathematically (just as Story did in his 2003 paper [9]) as:

$$E[\frac{V}{R}|R > 0] \times Pr(R > 0) \tag{1}$$

This mathematical definition explains what we claimed above, which is that the procedure controls the proportion of V Type-1 errors in the entire set R of errors $(R = S + V)$. The E term is expectation, and the Pr is the probability. The procedure itself operates as follows:

1. Sort the *p-values* in ascending order.
2. Find the largest index i in the set of tests such that the *p-value* at that index satisfies this equation:
   $$P_i \leq \alpha \times \frac{i}{m} \tag{2}$$
   where m = number of tests, i = index from [1, m], and α = the significance level.
3. All tests from index 1 to the resulting index from step 2 are then labeled significant.

This procedure, which was a breakthrough in the field of statistics, has a few faults. The first of which is, namely, the data set cannot be organized into different segments while performing the procedure, otherwise the results that are returned will be different than the results that returned from a non-segmented Benjamini-Hochberg procedure [7]. Separating the data into different sections may be useful for either organization purposes or to increase computational efficiency. In addition, sorting operations have a linearithmic time complexity. Therefore, in large-scale data sets (data sets on the order to 10^7 or higher), the sorting operation extends computational complexity. As a result, improvements needed to be made to the method such that the results can be compiled more efficiently for end-users.

2.2. Family-Wise Error Rate

There have been other solutions to the problem, most notably the *Family-Wise Error Rate* (FWER). FWER is the probability of there being at least one Type-1 error among all the hypotheses. Generally, FWER is very stringent on limiting the false discoveries, but in certain fields (i.e. genomics and/or bioinformatics) this stringency may not be necessary (when dealing with microarray experiments in the field of molecular biology, to give an example). In addition, FWER controlling procedures (such as the Bonferroni approach) have less *power* than that of FDR controlling procedures. The *power* of a procedure in statistics is the probability of rejecting the null hypothesis when there is an alternative hypothesis to the null hypothesis (and assuming that the alternative hypothesis is true). The Bonferroni approach [11] uses the following method (m is the number of hypotheses, and α is the significance level):

1. Sort the *p-values* in ascending order.
2. For every test P_i, for i in range $[0, m)$ reject all tests that satisfy the equation:

$$P_i < \frac{\alpha}{m} \tag{3}$$

As mentioned above, the Bonferroni procedure has many of the same faults as the Linear Step Up procedure, compounded with the fact that it has less power than that of the Linear Step Up. In addition, this procedure also returns many Type II errors [8]. Type II errors can be thought of as “false negatives,” or in statistical terms the error when a null hypothesis is false and you fail to reject the false null hypothesis. Controlling the Type-1 error rate should not inflate the Type-II error rate. Therefore, this model does not fit all of a researcher’s needs.

2.3. pFDR (positive FDR)

Another option that is similar to FDR is pFDR, or *positive* false discovery rate. This was proposed by Story in 2003 [9], and it is a slight modification on the original FDR controlling procedure proposed by Benjamini and Hochberg. Here, the procedure controls the probability that the null hypothesis is true even though it may have already been rejected. Mathematically it is very similar to the FDR, except the expectation is not multiplied by the probability that the number of tests is greater than zero. The full details of pFDR are described in Story's paper, but essentially omitting the probability in this procedure helps calculate an appropriate significance level, but overall it is related to the controlling procedure of the false discovery rate [9].

Therefore, the major goal of this project was to implement a new algorithm that improves upon the Linear Step Up procedure by decreasing runtime complexity while also limiting memory usage and addressing the shortcomings of previous solutions to the multiple comparisons problem.

3. FastLSU

In order to improve upon the method outlined in Benjamini and Hochberg, a novel approach to the Linear Step Up procedure was outlined by Madar and Batista. Rather than perform a linearithmic sorting operation, the new method, called FastLSU, performs linear scans over the data set using the following method [7]:

1. Count all p-values $< \alpha$, and store the result as c_1, where the "1" is the step number that ranges from 1 to k.
2. Count all p-values $< c_{k-1} \times \frac{\alpha}{m}$, and store the result as c_2. Here, m is the total number of tests being performed.
3. Repeat Step 2 (denote this step as the $k+1step$) until $c_k = c_{k-1}$ or when $k = m$, where k is the step number. These are the significant p-values.

As outlined in the original paper, this method generates the same number of significant results as the original method proposed by Benjamini and Hochberg, but when performing *large-scale* hypothesis testing, the performance of FastLSU scales more reasonably because it is linear as opposed to linearithmic. Furthermore, as noted in Section 7.1, the results from FastLSU are equivalent to that of the more traditional Linear Step Up procedure. As described in Section 4.1, the specific implementation actually segments the data into different chunks based on the number of available cores in a given machine. The FastLSU method varies slightly when it uses segmented data. The modified procedure is illustrated below, where n is the number of cores [7]:

1. For each segment, count the number of p-values $< \alpha$, storing the result as $r^1_{c_i}$.
2. For every segment, count the number of p-values $< (\sum_{i=1}^{n} r^1_{c_i}) \times \frac{\alpha}{m}$, and store the result as $r^{k+1}_{c_i}$, where k is the step number.
3. Repeat Step 2 (denote this step as the $k+1 step$) until $\sum_{i=1}^{n} r^k_{c_i} = \sum_{i=1}^{n} r^{k+1}_{c_i}$ or when $k = m$, where k is the step number. These are the significant p-values.

It is important to note that in this procedure, the segments are processed synchronously. This is to say, each segment must finish Step 1 before any segment can start Step 2 (this is because, Step 2 requires the sum of all the results from Step 1 in order to do the comparison in Step 2). Likewise, all segments must complete Step 2 before a single segment can begin Step 3 (similarly, Step 3 requires a comparison that involves the sum of the results from Step 2). Figure 1 illustrates this syncing mechanism. The RM is the Resource Monitor, an abstraction that keeps track of the counts from each chunk for every full iteration of the procedure. Once a step finishes for all the segments, each segment sends the Resource Monitor its count, which tallies it into a sum to be used in the comparison for the next step.

The benchmarks for the implementation are outlined in Section 7.3. What distinguishes the FastLSU from the other algorithms is that is more *powerful* than the other methods of FWER and Bonferroni. By more *powerful*, this is to say that the FastLSU method has a greater ability to control the rate of Type 1 errors than other methods [7]. Furthermore, the algorithm overall has a linear

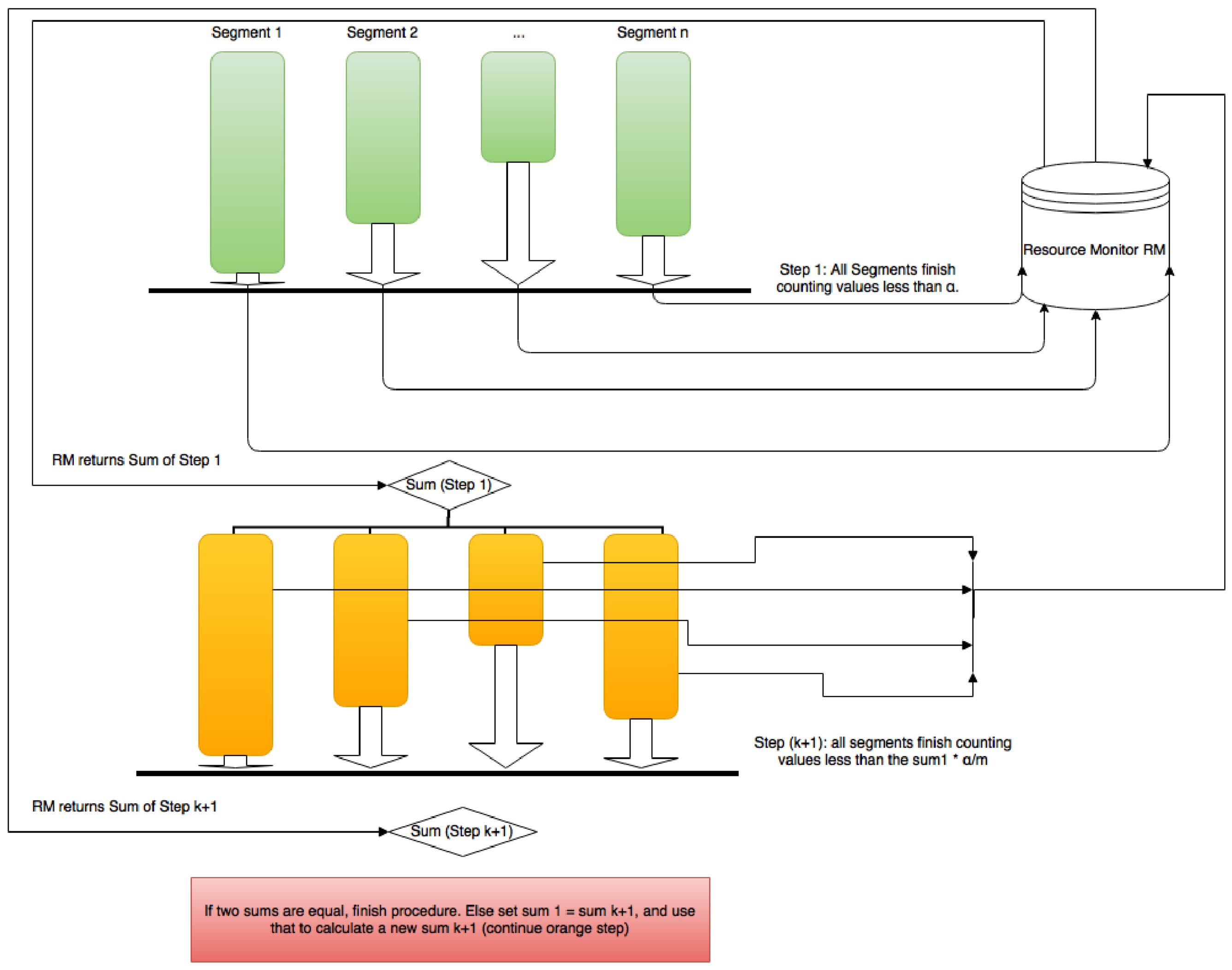

Segment 1
Segment 2
...
Segment n
Resource Monitor RM
Step 1: All Segments finish counting values less than α.
RM returns Sum of Step 1
Sum (Step 1)
Step (k+1): all segments finish counting values less than the sum1 * α/m
RM returns Sum of Step k+1
Sum (Step k+1)
If two sums are equal, finish procedure. Else set sum 1 = sum k+1, and use that to calculate a new sum k+1 (continue orange step)

Figure 1: Resource Monitor Diagram. The two diamonds represent the two sums, and if they are not equal, then the first sum gets set to equal the second, the the flow of the procedure continues from the yellow portion down to the red.

time complexity. To illustrate this, let us consider the worst case-scenario: image if only one test gets rejected during every iteration (if no tests get rejected, we are finished, and if more than one gets rejected, the final number of rejected tests will converge to a value more quickly). Suppose we are looking at an experiment with only 10 tests, and there is only one segment. If we only reject one test per iteration until we reached zero, then the number of tests we observed would have been 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55. Thus, we would have rejected at most *half* of the total number of tests. We can visually represent this as a grid as in Figure 2, where the white squares are the rejected tests for a given row, and the red squares are what we are scanning as we move down row-by-row. Even though we did a total of ten scans, the number of elements that were not rejected decreased by one. Therefore, the running time would be $\frac{N}{2}$, or $O(N)$, where N is the number of tests.

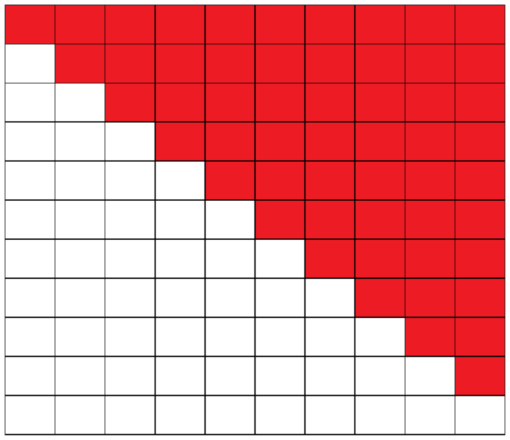


Figure 2: Visual representation of linear time complexity.

The challenge for this project then lied in translating and implementing the algorithm effectively for use by researchers and statisticians, leading to import design decisions that needed to be considered. In particular, the implementation had two overall goals: to minimize both the run-time and minimize memory usage on machines. In order to accomplish this, important design decisions had to be made.

3.1. Memory Constraints

There may exist certain cases that may not fit into system memory (i.e. the file size of the experiment may exceed the practical RAM limits for a particular machine). In order to resolve this problem, we followed a different procedure when processing the results – this procedure is based on Algorithm 1 (see here -> 3). Let m^* be the amount of tests that can fit in memory, m be the total number of tests, t be the number of remaining tests and α be the significance level.

1. Load the $\min(m^*, t)$ number of tests into memory. Let r be the result of the minimum function.
2. Segment the r tests into n different segments, based on the number of cores available. Let S_i

be the size of each segment.

3. In each segment, count the number of tests less than α. Denote this is as c_i^1 (note $k = 1$).
4. In each segment, count the number of tests less than $(m - S_i + c_i^1) \times \frac{\alpha}{m}$. Denote this as c_i^{k+1}.
5. In each segment, repeat step 4 until $c_i^{k+1} = c_i^k$. When doing step 4, the condition changes to $(m - S_i + c_i^{k+1}) \times \frac{\alpha}{m}$ in each iteration. Save the results from each segment into one temporary file.
6. Repeat steps 1-5 for the remaining tests in file, making sure to save the final results into the *same* temporary file.
7. Finally, perform Algorithm 1 on this temporary file, still using the same m value (m being the total number of tests).

Figure 3 is a flowchart for this particular procedure. Since this procedure is segmented based on the number of available cores, we took advantage of concurrency when multiple cores were present. With a low α value, as long as the temporary file is relatively small, this final step can be completed rapidly.

4. Design Decisions

4.1. Design Outline

In order to implement the algorithm, it was first necessary to decide which technologies to take advantage of so that the overall goals could be accomplished. As outlined above, a major component of the FastLSU algorithm involved being able to arbitrarily segment the data and still return the same results as if one worked on an entire batch of data. This segmentation, or *chunking*, as it is referred to in the FastLSU paper, therefore proved to be useful because segmentation would allow for parallel processing on multi-core machines (the vast majority of stock machines have many cores, unless the algorithm is being run under a virtual machine). Therefore, the implementation needed to be designed such that the input data could be divided up between the number of available cores, and then having each core process a segment of the data in parallel in order to get the final

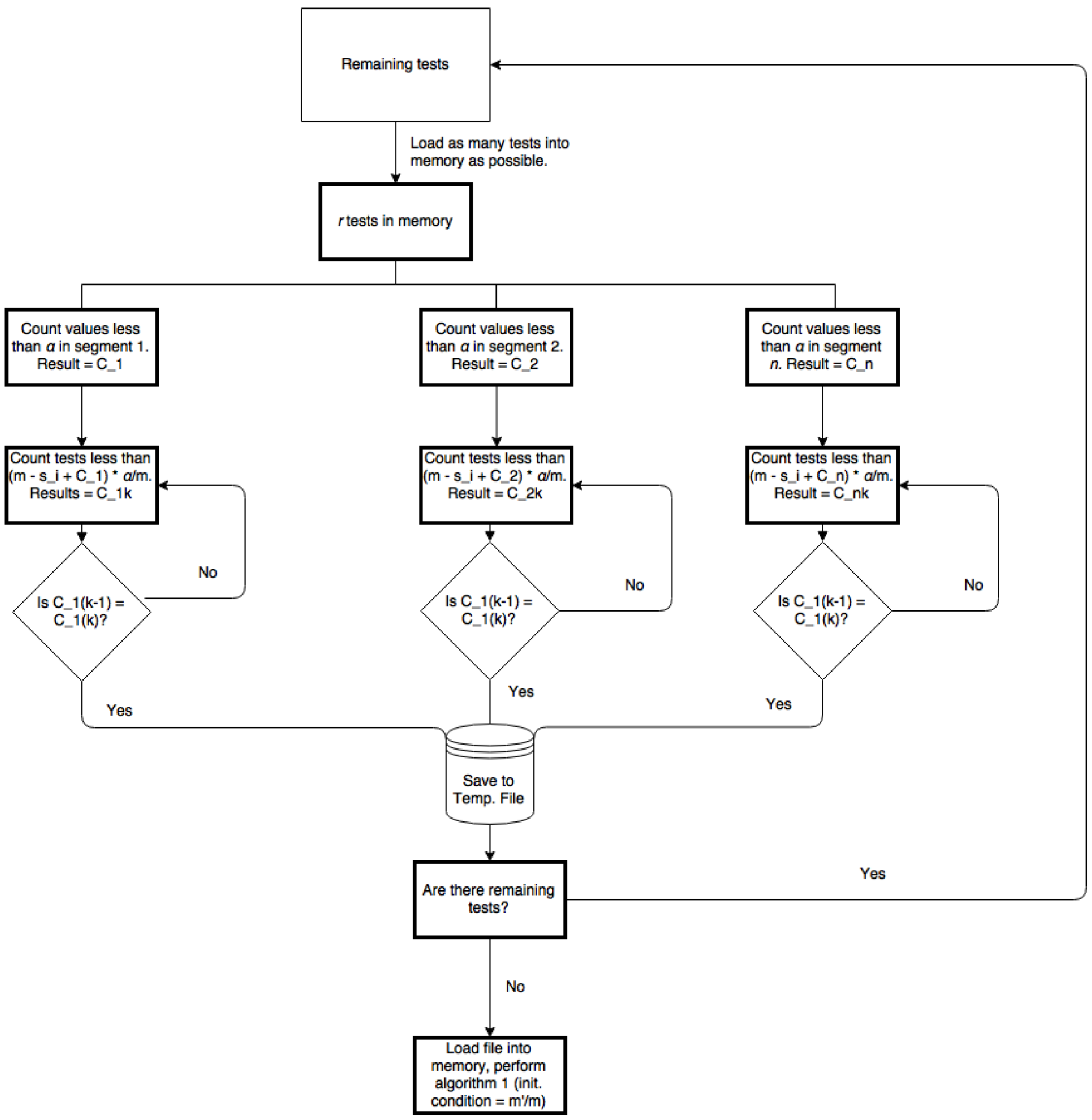

Remaining tests
Load as many tests into memory as possible.
r tests in memory
Count values less than α in segment 1. Result = C_1
Count values less than α in segment 2. Result = C_2
Count values less than α in segment n. Result = C_n
Count tests less than (m - s_i + C_1) * α/m. Results = C_1k
Count tests less than (m - s_i + C_2) * α/m. Result = C_2k
Count tests less than (m - s_i + C_n) * α/m. Result = C_nk
Is C_1(k-1) = C_1(k)?
No
Yes
Save to Temp. File
Are there remaining tests?
Yes
No
Load file into memory, perform algorithm 1 (init. condition = m'/m)

Figure 3: Workflow for Algorithm 2

result.

Algorithm 2 (found here -> 3) was better suited this general method. Since the algorithm is able to perform large-scale hypothesis testing with arbitrary segments, FastLSU in a multi-core environment works best with Algorithm 2, which is designed to be utilized with arbitrarily-sized segments (in this case, the number of segments would just equal the number of available cores). However, step $k+1$ of that algorithm requires that every p-value in a particular segment has to be compared with an inequality that contains the sum of the previous steps' counts. Therefore, in order to find the sums across all the cores, all that was required was a thread-communication mechanism to report the sum across the segments. As the counts were calculated, a table in memory gets update with the segment ID, the current step number, and the count at said step number. Once all the entries in the table achieve the same step number, return the sum across all the steps. Once this part was achieved, then the rest of the implementation could focus on improvements and actually implementing the algorithm.

Overall, implementation of the method required multiple design decisions and challenges. These were:

- Choosing an implementation language
- Deciding with method from the FastLSU paper (Algorithm 1 or Algorithm 2) would minimize both space usage and runtime.
- Determining an efficient segmentation method for Algorithm 2.
- Have a mechanism to keep track of the sums between segments.
- Decreasing the load time for large-scale hypothesis testing.
- Take advantage of available system resources (including number of cores and available memory).
- Present an easy-to-use interface that allows for simple usage without resorting to command-line prompts.
- Ultimately, allow the implementation to be easily modifiable by future programmers interested in the task; i.e. the implementation must serve as a living project.

Each one of these points led to particular decisions based on the design goals that were outlined prior to the completion of the project.

One of the earlier design decisions involved choosing which programming language would be best suited for this kind of project, taking into account the different tradeoffs that needed to be made. Because a major goal of this project was to decrease run-time, it was decided that a fully interpreted language would not meet our needs (interpreted languages such as Python tend to have slower I/O access times – this is not to rule out a potential Python wrapper in the future). Ultimately, Java was chosen because of its relatively speedy I/O times when loading in data, its libraries for performing concurrent tasks as well as being programmer-friendly enough so that future undergraduates/open-source programmers can add extra functionality more easily over the course of time. Furthermore, Java utilizes a garbage collector that handles memory management for the developer. Thus, Java allowed us to only have to worry about the maximum memory we decided to use for the application; memory management was handled for us.

Next it had to be decided how to specifically segment the data into different segments of close to equal size (so that no core carries an undue burden of the processing over another core). I used the following method to segment the data (let n be the number of total tests, and m be the number of available cores):

1. For every index i in range [0, number of cores), let the starting index of segment$_i$ be the floor$(n \times \frac{i}{m})$, and let the end index of segment$_i$ be the floor$((n \times \frac{i+1}{m}) - 1)$.
2. Repeat step 1 for all of the available segments.

Segmenting in this number minimizes the size of each segment. To illustrate, consider an example in which there are four available cores $(m = 4)$ and ten tests $(n = 10)$. For core 1, the indices would be [0, 1], core 2 would be in the range of [2, 4], core 3 would be in the fange of [5, 6], and core 4 would handle tests [7, 9]. Once the input data is segmented, the tests are then loaded into memory and then processed concurrently.

To accomplish our final goal, of making this tool readily available and updatable, we chose to use well-documented technologies such as JavaFX (for graphical user interface creation [2]) and the Java concurrency packages. As an open-source project, the source code will ultimately rest on platforms such as GitHub [4] in order to encourage others to update the code-base with new features (such as performance improvements and/or GUI features that allow for multiple *experiments* to run simultaneously).

5. Implementation Details

5.1. Java classes

A few classes needed to be defined in order to improve efficiency. The classes defined in this implementation are as follows:

- FastBHConcurrent: a main processing class that segments data into appropriately-sized segments based on the number of cores available.
- ResourceMonitor: a class that keeps track of the sum of every individual step for all the segments (see Figure 1).
- WorkerThread: a class that is instantiated *x* number of times (with *x* being the number of cores available) – this performs the actual Fast Linear Step Up procedure.
- MemoryThread: this class is used in lieu of a WorkerThread if the data cannot all fit into memory.
- PValues: a class that acts as a tuple in Java. Here, this class holds the data for a particular segment as well as the number of tests within that chunk that were less than the inequality in algorithm 2 in the final iteration.
- QV: a class that can calculate "adjusted p-values" – *q-values* – for the results of FastLSU (see Section 8.1).

The ResourceMonitor is based on the ConcurrentHashMap data type from the Java SDK (Standard Development Kit) – this allows for concurrent reads in the map (in our case a symbol table of keys being thread names and values being the count of the particular segment) and only locks a particular portion of the map when writes occur. This is done for thread safety reasons as well as avoiding atmocity errors.

Note that for experiments that may not fit into memory, disk access is required. Determining the maximum number of tests involved checking the amount of memory allocated by the Java Virtual Machine (JVM). This provided a reasonable estimate to the maximum number of tests in memory. In either scenario, the same result is produced, albeit the out-of-memory tests use a slightly modified procedure than that of FastLSU proper, outlined in Figure 3. To further optimize the implementation, during I/O access, Step 1 of the FastLSU procedure (for the second algorithm - click here -> 3) is performed. This was done to lessen the burden on each concurrent thread.

5.2. Data Types

The individual *p-values* themselves are stored in arrays as opposed to other data structures such as ArrayLists or ArrayLists. As a primitive data type, arrays have very fast read times as well as insertion time (assuming re-allocation is not necessary). While array removal has a linear cost, rather than do this, for every p-value that failed the comparison, the array index was set to -1. In the following table, tests were done to decide whether or not to use ArrayLists, HashMaps, or primitive arrays as the base data structure for holding all of the hypothesis. Other data types such as LinkedLists, Stacks, Sets, and Queues were ignored for various reasons: LinkedLists had performance similar to that of ArrayLists, Sets do not allow duplicate tests, and Stacks and Queues are implemented based on List data structures.

Table 2: Runtimes for different data types on test of size 1.0×10^7

Data Type	Avg. Load Time (ms)	Avg. Proc. Time (ms)	1st Load Time (ms)	1st Proc. Time (ms)
ArrayList	5348.0	4344.8	7845.0	9239.0
HashMap	12970.4	4965.0	13743.0	5084.0
Array	2820.2	1477.8	3027.0	1500.0

Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM, Segments = 1

In these test cases, the number of segments was set to 1. The table shows that primitive arrays have demonstrate decrease in average reading in time compared to the other two. Since in the

Array and ArrayList rows we did not actually remove any p-values (removing was necessary for the HashMap), these two rows show a decrease in processing time as well. In addition, out of all these data structures, arrays have the least memory overhead of them all, and furthermore do not require an additional function (i.e. hash functions for the HashMap) when doing lookups, thus runtime for arrays tends to be smaller than that of HashMap for our purposes.

5.3. Design Trade-Offs

The design decisions did not come without a few compromises that needed to be made. Primarily, the JIT ("Just-In-Time") compiler within the Java SDK. The JIT allows for compiler optimizations to be made during the runtime of an application. This process impacts runtime for the first iteration of an algorithm – in Java parlance, JIT optimization occurs during the JVM warm-up time . In looking at the average runtime of the algorithm (see Section 2), we noticed that the first run took more time to complete than subsequent iterations. Still, the runtime was ultimately much smaller than that of the slow version, and as such the choice of Java was solidified. A possible solution in future would be to run this tool on a server continuously – this would eliminate the penalty for the JIT compiler. In our GUI tool, to address this problem, we perform the FastLSU procedure on 1 million random, uniform values in the range [0,1) prior to the application loading. This helps limit the runtime when *processing* data. However, we still run into the issue of JIT and how it can optimize load-times. One might consider bundling our tool with a sample experiment in file, but for users who would want to run our tool using the command-line (if they download the binaries only) this option might not be optimal for them. Nevertheless, we were able to optimize some parts of the JIT in the implementation. In particular, we used the *final* keyword when defining many methods in Java. The *final* keyword helps the JIT compiler because the compiler will not have to lookup a method and its inheritance table to find the base method that is being called. With more methods defined and the warm-up, we decreased JIT time.

Furthermore, other languages such as C/C++ can have faster I/O times when loading data [1] than Java, as well as having concurrency libraries that can outperform some of the Java concurrency packages. However, for the scope of this project, it was not essentially to create the tool in this language. Since many machines may have 1-4 cores, there are not too many threads being created – as such, a more complicated multi-threaded environment seemed unnecessary. Limiting I/O was another major design goal. To minimize I/O times, any disk reads/write were done sequentially (concurrent disk reads severely impact performance as multiple seeks need to be performed).

Another import decision that was made was determining the maximum amount of RAM this program would use. While the JVM does initialize itself with a certain amount of system memory, this may not be enough/may be too much for a particular machine. Therefore, it was decided that the implementation would not use more than 75% of the available memory to a machine. In order to accomplish this, we created a Windows executable on top of an executable JAR file (JAR files are packages of compiled Java code that can be executed with a double click), and after that we used the launch4j tool [6] to both create the Windows binary and limit the maximum memory usage to the 75% threshold. This is not to say that the application always used 75% of available memory – rather, this is the maximum possible that is allotted. Even so, in the implementation, when calculating the largest array that can fit inside of memory, we estimate a value that is lower than the available memory. For example, if the maximum heap size is set to 3 GB, we limit the array size to 2.25 GB (it is always safer to be conservative). Furthermore, actually estimating the largest array size that can fit in a particular JVM instance was challenging – we found that, experimentally, finding the maximum memory alloted to a JVM instance, dividing that by 10, and then removing 10% of that result (in order to be a bit more conservative and not directly hit or limit) gave us a reasonable estimate into the maximum number of tests we could load in memory. Every machine may vary slightly when it comes to the maximum allotted memory for the JVM - furthermore, if a 32-bit JVM is installed as opposed to a 64-bit version, then the largest number of tests that can be processed in memory would be limited.

We assume that the data the user input is properly formatted. This would mean that the data is either space or tab delimited. Initially we intended to use the regular expression package in java to split strings by whitespace, but we found that using regular expressions increased load times, so instead we implemented a simple String tokenizer that utilizes switch and case logic to separate a string into its appropriate values. If we compare the times in Table 3, we see that regular expressions (the regular expression used was "$\backslash\backslash s$", which handles all whitespaces and tabs) perform worse than the custom method. If the input was perfectly formatted such that only one space character delimited every test, then this would give the fastest load times. This was done by using the Java "split" method with a single space character. However, in order to be flexible to other input types (such as the researcher putting extra spaces or using tabs and/or spaces), we decided to use the custom method. The average load times overall for a variety of tests (see Figure 11) were still reasonably low. Later on, since we have to parse the Strings as doubles, we take care of the problem in which the user might give improper input (such as words between the tests or things of that nature). We accomplish this by catching any error that the parse function in Java throws.

Table 3: String tokenization methods on input of size 1.0×10^7

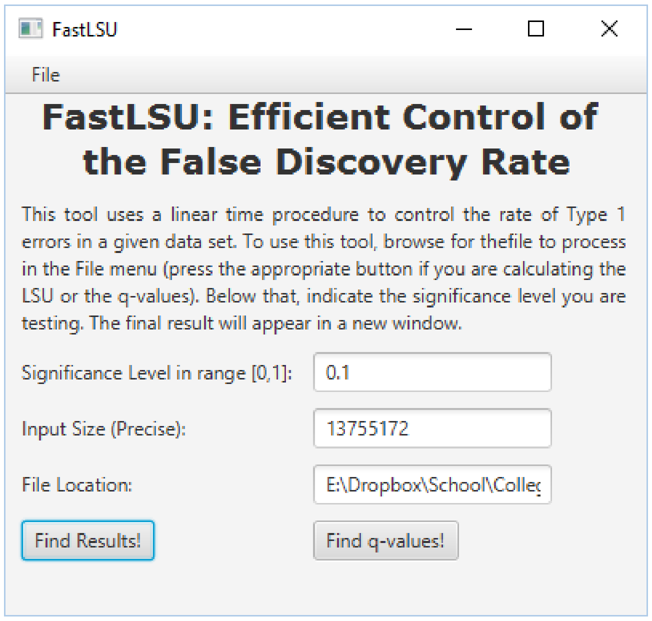
Type	Time (ms)
Regular Expression	2014.2
Custom Tokenizer	1565.6
Single Char. Tokenizer	1160.4

Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

6. Simple Graphical User Interface for the FastLSU Procedure

In order to make this project usable, we created a simple GUI application that researchers can use to perform both the base FastLSU procedure as well as calculating q-values post-processing. The main screen of the tool can be seen in Figure 4. This allows a user to input the total number of tests as well as the significance level and the file location and generate a subset of results.

Pressing either the "Find Results!" or the "Find q-values" will perform the FastLSU or the q-value procedure, respectively. The results will be tabulated in a new window, in which the user can choose to export the data into a text file. We intend to also allow exporting to CSV files and other data formats as well, including direct exportation to the Excel program. The following figure (Figure



FastLSU: Efficient Control of the False Discovery Rate

This tool uses a linear time procedure to control the rate of Type 1 errors in a given data set. To use this tool, browse for thefile to process in the File menu (press the appropriate button if you are calculating the LSU or the q-values). Below that, indicate the significance level you are testing. The final result will appear in a new window.

Significance Level in range [0,1]: 0.1

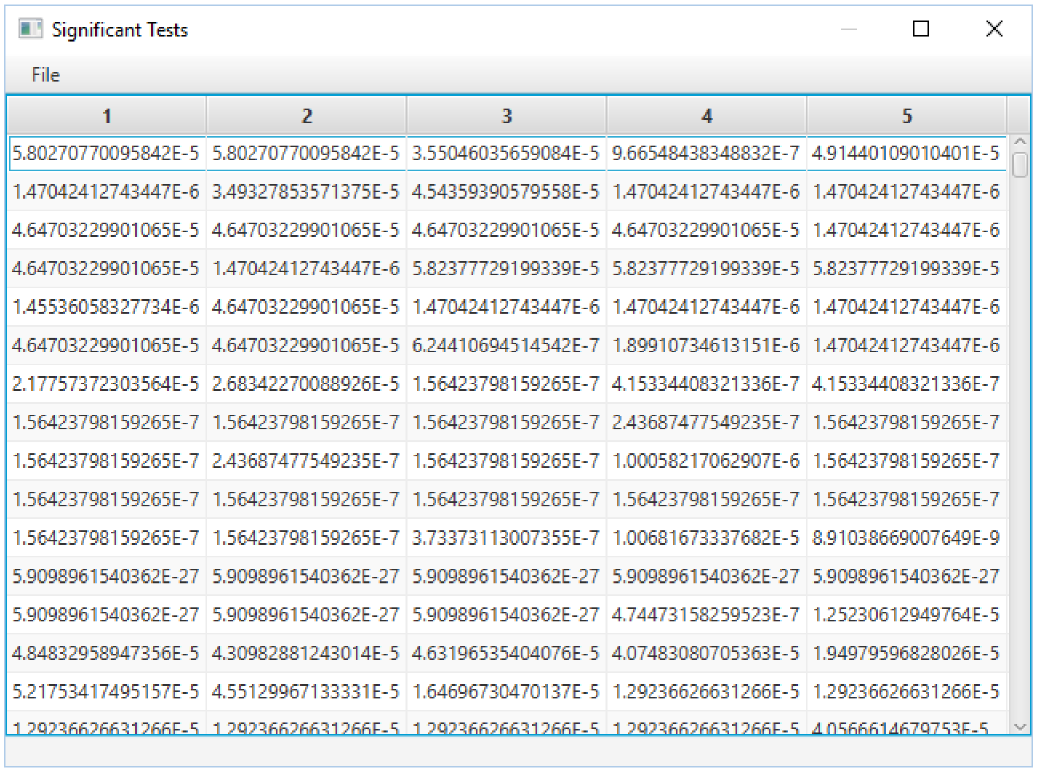
Input Size (Precise): 13755172

File Location: E:\Dropbox\School\Colleç

Figure 4: Splash screen for our GUI. The user inputs a significance level, the number of tests, and the file location.

5)highlights the results from Stranger's [10] cis-eQTL study for the Central European HapMap group (see Section 7.1). It is also user-friendly to calculate the q-values from the results screen of the significant results: the command is in the File menu. Furthermore, if a researcher already had results stored in file and needed to calculate the q-values, then the GUI tool allows them to simply load in the results and quickly calculate the q-values for the results – from there the researcher can save them to file and utilize them as they wish.

In order to create this tool, we used the JavaFX package that is bundled inside of the Java 8 SDK. This package in particular is very flexible and takes advantage of XML and CSS when customizing the look and feel of GUIs, in combination with Java source code. There are many tools and IDEs (Integrated Development Engines) such as NetBeans and Eclipse that help developers create GUIs more efficiently. Thus, we chose to use this framework in order to make it easier/quicker to make adjustments to the look and feel of the tool. Furthermore, for developers in the future who wish to customize/add new features to our tool, having a simple framework makes the most sense when encouraging others to contribute to the project.



1	2	3	4	5
5.80270770095842E-5	5.80270770095842E-5	3.55046035659084E-5	9.66548438348832E-7	4.91440109010401E-5
1.47042412743447E-6	3.49327853571375E-5	4.54359390579558E-5	1.47042412743447E-6	1.47042412743447E-6
4.64703229901065E-5	4.64703229901065E-5	4.64703229901065E-5	4.64703229901065E-5	1.47042412743447E-6
4.64703229901065E-5	1.47042412743447E-6	5.82377729199339E-5	5.82377729199339E-5	5.82377729199339E-5
1.45536058327734E-6	4.64703229901065E-5	1.47042412743447E-6	1.47042412743447E-6	1.47042412743447E-6
4.64703229901065E-5	4.64703229901065E-5	6.24410694514542E-7	1.89910734613151E-6	1.47042412743447E-6
2.17757372303564E-5	2.68342270088926E-5	1.56423798159265E-7	4.15334408321336E-7	4.15334408321336E-7
1.56423798159265E-7	1.56423798159265E-7	1.56423798159265E-7	2.43687477549235E-7	1.56423798159265E-7
1.56423798159265E-7	2.43687477549235E-7	1.56423798159265E-7	1.00058217062907E-6	1.56423798159265E-7
1.56423798159265E-7	1.56423798159265E-7	1.56423798159265E-7	1.56423798159265E-7	1.56423798159265E-7
1.56423798159265E-7	1.56423798159265E-7	3.73373113007355E-7	1.00681673337682E-5	8.91038669007649E-9
5.9098961540362E-27	5.9098961540362E-27	5.9098961540362E-27	5.9098961540362E-27	5.9098961540362E-27
5.9098961540362E-27	5.9098961540362E-27	5.9098961540362E-27	4.74473158259523E-7	1.25230612949764E-5
4.84832958947356E-5	4.30982881243014E-5	4.63196535404076E-5	4.07483080705363E-5	1.94979596828026E-5
5.21753417495157E-5	4.55129967133331E-5	1.64696730470137E-5	1.29236626631266E-5	1.29236626631266E-5
1.29236626631266E-5	1.29236626631266E-5	1.29236626631266E-5	1.29236626631266E-5	4.05666146797537E-5

Figure 5: The results window from the GUI tool. The file menu allows for exportation to *.txt files, as well as the calculation of q-values.

7. Empirical Analysis

It was important to test both the runtime and the memory usage of our implementation in order to verify our claims that the runtime complexity is $O(m)$ and the space cost is also $O(m)$, where m is the number of tests.

7.1. Real-World Analysis

The project was first tested on the same real-world data taken from the different HapMap groups in Stranger's cis-eQTL experiment [10]. Tests on real-world data were necessary in order to show correctness. In addition, since real-world data may not necessarily be uniform, runtimes can vary depending upon the size of the data set. In the following table (Table 4), we compare the initial runtimes of the traditional Linear Step Up to the FastLSU procedure. In this experiment, the significance level was set to $\alpha = 0.1$. Each group had approximately 14 million hypotheses to test, and we used four segments for the FastLSU.

HapMap group	LSU read (ms)	LSU proc. (ms)	FastLSU read (ms)	FastLSU proc. (ms)
Central Europeans	6268.0	2304.0	6386.0	306.0
Chinese and Japanese	6172.0	2110.0	8214.0	251.0
Nigerians	6883.0	2455.0	7543.0	242.0

Table 4: Initial runtimes for Fast & Naive LSU on the HapMap groups. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

In both cases, each procedure returned the precise same results: 9,228 from the Central European group, 33,507 from the Chinese & Japanese phenotype, and 6,497 from the Nigerian population. Even when taking into account the I/O times as well as initialization time, the FastLSU algorithm still produces an improvement in speed compared to the traditional Linear Step Up. Figure 6 shows the average runtime analysis for each of the HapMap groups. We were able to achieve near-equal disk access times while also decreasing the overall runtime of the application significantly. When compared with the runtimes for the random uniform data, we see that the runtimes are slightly different for the same order of magnitude (10^7). We claim that this is due to a few things: first, the real-world data has approximately ~3 million more tests than that of the random uniform data and secondly, the real-world data may be compiled such that more tests are considered every iteration (there may be a lot of false positives in the experiment).

7.2. Memory Profiling Results

The FastLSU procedure uses $O(m)$ space (m = number of tests). This is due to the fact that once all the tests are stored in memory (if the experiment can fit) no more data structures are created. In the workflow outlined in Section 3 (Figure 1), we see that after the data is loaded in, it is processed in place, and finally the results are returned as a list of values. It is important to profile so that one can visualize the memory usage of the application. To do this, the Java VisualVM memory profiler was used [3]. This tool allowed for the viewing of the memory profile (how much memory the application is using) as well as the proportion of time spent in any particular method for any

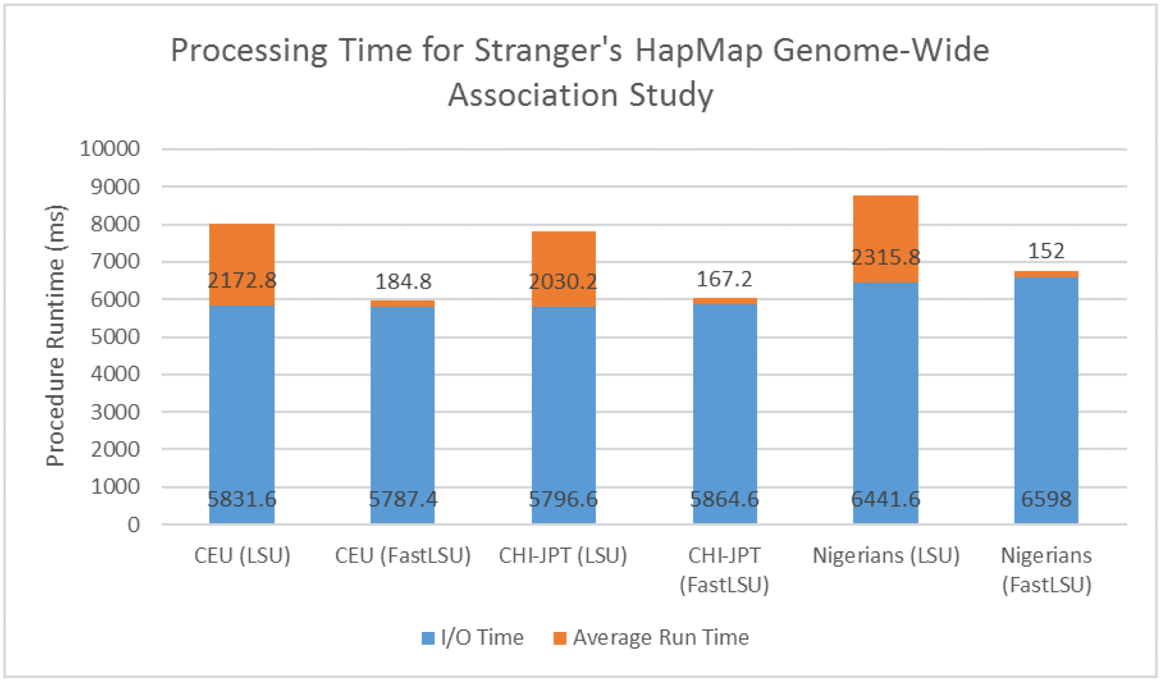
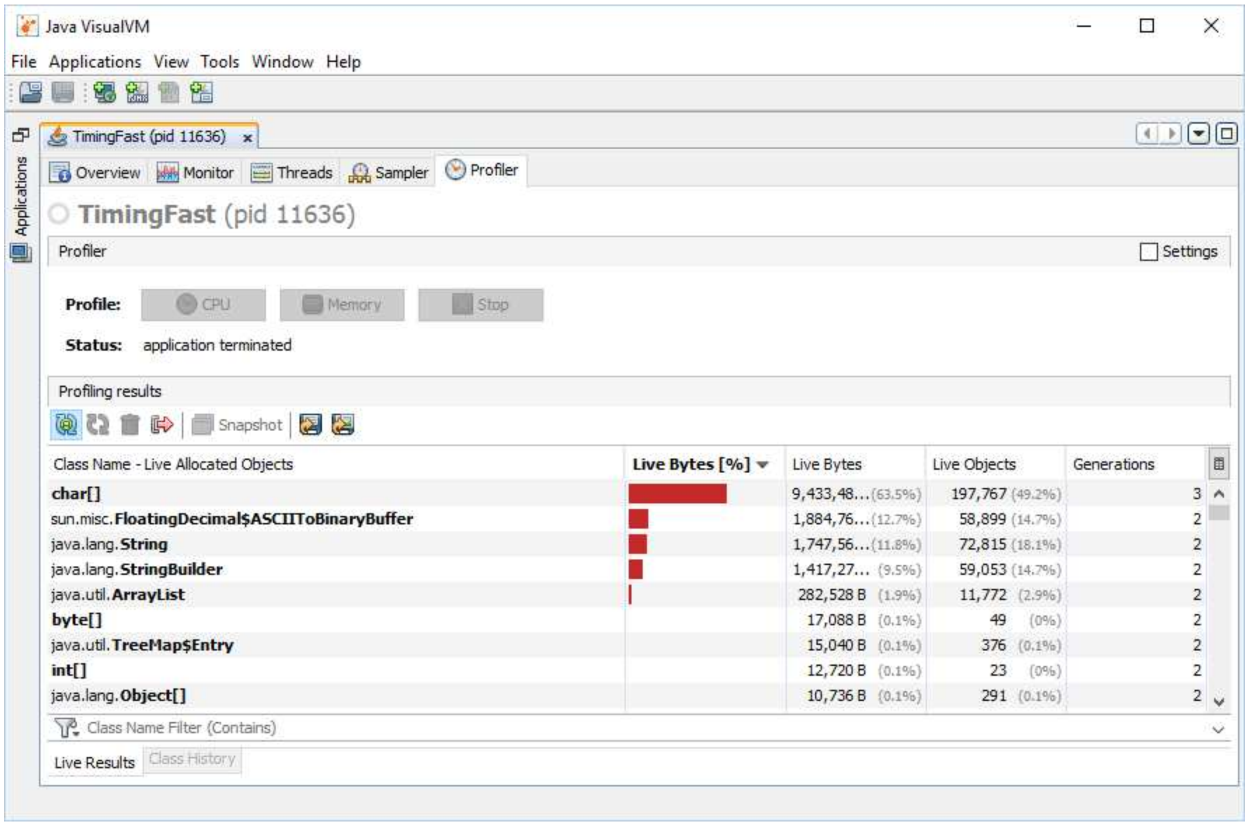


Figure 6: The different HapMap groups and there respective runtimes for LSU and FastLSU - load times are in blue, procedural times are in orange. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM. Segments = 4.

particular segment.

The more important fields to consider were the array of p-values (represented as the double primitive), the ResourceMonitor itself, and any small overhead required for class definitions and the like. The goal of memory profiling was to show that, indeed, the amount of memory used by the program never exceeded our pre-defined limits. The ResourceMonitor itself took up a neglible amount of bytes (nearly zero). This was expected because it simply holds the counts for all segments; the maximum size of the ResourceMonitor is bounded by the number of segments. Since the PValues class's underlying data structure is a double array, we saw that the memory profile was largest for the double array portion. What initially surprised us was the amount of memory that Strings ultimately use. The underlying String primitive, char arrays, also had a decent amount of bytes allocated. The following tables outline the total memory usage of the implementation. In the first few lines, we see the memory usage for an experiment with exactly 10^8 hypotheses and a significance level (α) of 0.1.


Java VisualVM
File Applications View Tools Window Help
TimingFast (pid 11636)
Overview Monitor Threads Sampler Profiler
Applications
Settings
CPU Memory Stop
Snapshot
Class Name Filter (Contains)
Live Results Class History

TimingFast (pid 11636)

Profiler

Profile:

Status: application terminated

Profiling results

Class Name - Live Allocated Objects	Live Bytes [%]	Live Bytes	Live Objects	Generations
char[]	🟥	9,433,48... (63.5%)	197,767 (49.2%)	3
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	🟥	1,884,76... (12.7%)	58,899 (14.7%)	2
java.lang.String	🟥	1,747,56... (11.8%)	72,815 (18.1%)	2
java.lang.StringBuilder	🟥	1,417,27... (9.5%)	59,053 (14.7%)	2
java.util.ArrayList	🟥	282,528 B (1.9%)	11,772 (2.9%)	2
byte[]		17,088 B (0.1%)	49 (0%)	2
java.util.TreeMap$Entry		15,040 B (0.1%)	376 (0.1%)	2
int[]		12,720 B (0.1%)	23 (0%)	2
java.lang.Object[]		10,736 B (0.1%)	291 (0.1%)	2

Figure 7: Sample screen in which the tests are still being loaded into memory. Number of segments = 4, Hardware = Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

Class Name - Live Allocated Objects	Live Bytes	Live Objects	Allocated Objects
double[]	200,000,016	1	1
char[]	17,644,560	369308	5,775,777
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	3,684,832	115,151	1,803,611
java.lang.String	3,312,504	138,021	2,164,697
java.lang.StringBuilder	2,769,600	115,400	1,803,622
java.util.ArrayList	552,768	23,032	360,930

Table 5: Memory profile for a specific segment during processing. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM, Segments = 4

Class Name - Live Allocated Objects	Bytes Allocated [%]	Bytes Allocated
double[]	65.20%	800,000,712
char[]	16.88%	207,109,048
$ProfilerRuntimeObjLivenessWeakRef	3.96%	48,606,240
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	3.52%	43,183,072
java.lang.String	3.19%	39,100,344
java.lang.StringBuilder	2.64%	32,391,480
int[]	2.03%	24,947,336
java.lang.Object[]	1.25%	15,313,576
java.lang.ref.WeakReference[]	0.67%	8,224,784
java.util.ArrayList	0.53%	6,480,480

Table 6: Memory overview of total memory usage during processing+loading. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

Most of the memory used involved storing the tests themselves in memory (this was done using the double primitive). Both tables follow a general pattern in which double array(s) hog most of the memory, followed by a temporary char array (this was used during loading). The first table is a direct memory profile, while the second table looks at the code as a whole to determine memory usage. The number of bytes used fits well with the claim that the space costs are linear to the number of tests. We can observe that, while the test cases use the largest percentage of memory in the JVM, most of the other items are smaller header penalties or base classes that are extended by some of the inherent Java classes, including the ConcurrentHashMap that the ResourceMonitor was based on. As the number of tests were scaled, we saw that the memory usage scaled accordingly. The ResourceMonitor, however, did not use more memory between sample sizes. In the implementation, if any segment returned a final count of 0, we removed the tests in that segment from memory. Thus, in some cases, the memory usage can actually *decrease* with time.

The second table, meanwhile, is a sample of the memory usage at the end of the procedure (another way to think about Table 5 is the memory usage for one particular segment, while Table 6 is the overall memory usage for the entire application). As we expected, the double array utilizes most of the memory. The char array that uses memory here is actually just the buffer used by the BufferedReader class in java. This memory was used in order to decrease the read time from disk. As the procedure progressed, the amount of memory used by char arrays would decrease when the Java collector ran while the data was being processed. For multiple trials, we observed the memory usage of the char array increase and decrease. The graph in the Figure 8 illustrates the nature of the char array buffer. In this case, there were four segments. The circled regions in the graph show the points in which the char buffer was cleared from memory.

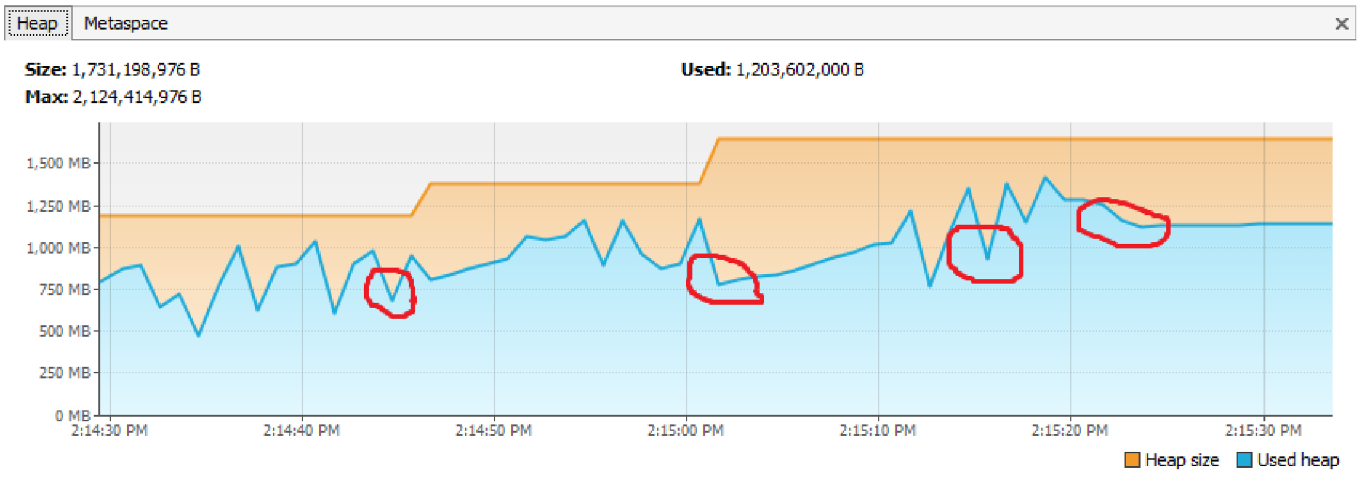


Figure 8: Memory usage for 10^8-sized experiment over time. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

Most of the other peaks and valleys throughout the runtime involve runs of the Java garbage collector over time. Fortunately, the garbage collector did not drastically affect the overall runtime of the tool. In addition, we actually do not attempt to reach the absolute maximum heap size. It is the orange line in Figure 8. We decided it was better to be more conservative when it came to our estimation for the number of tests that could fit in memory so that the tool overall could run smoothly with "room to spare." Not only would this allow for multi-tasking on the part of the user, but if our tool needed the extra memory to store the results before they are displayed on screen, then having the extra heap space proved to be beneficial.

7.3. Timing Results

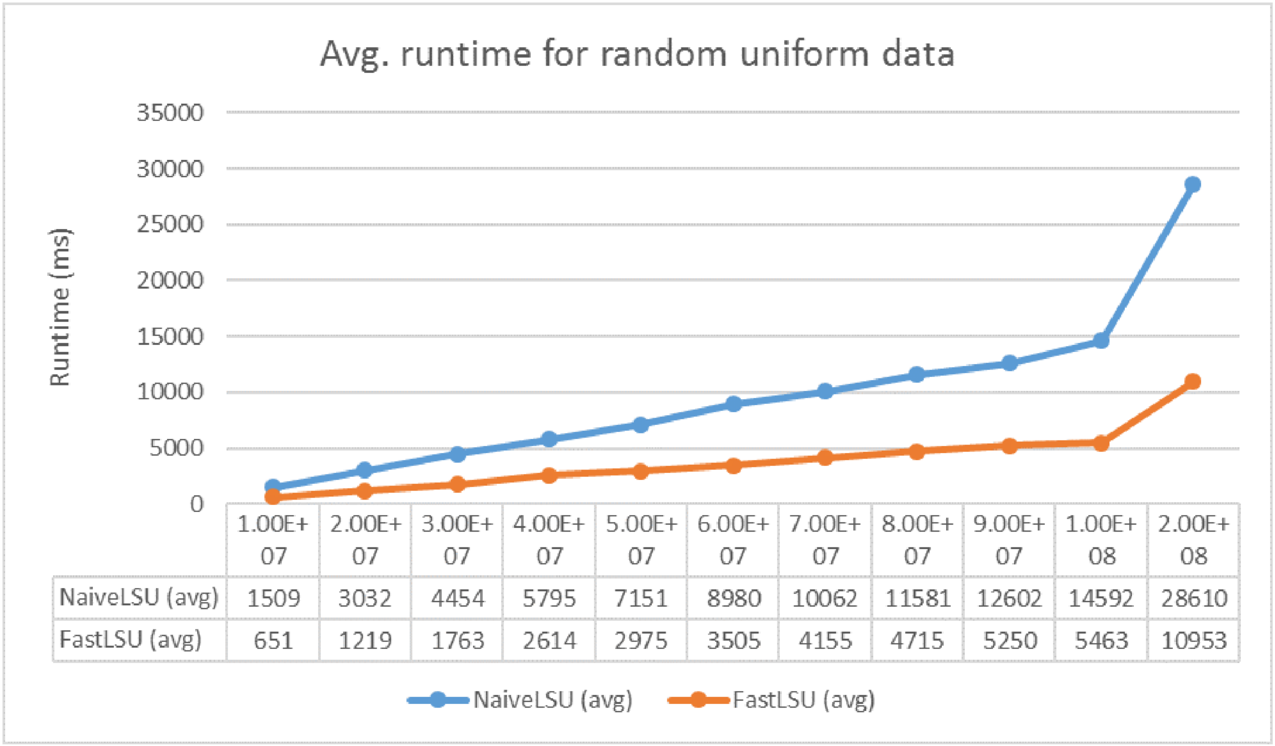
We have compiled timing results for a variety of test cases. For all the test values, we have generated random uniform p-values as well as a random binomial distribution of test cases. A random binomial distribution can be thought of as a series of coin flips that are independent of previous coin flips: the result can either be "tails" (a "0") or "heads" (a "1"), but the next coin flip's result will be *independent* of the previous result. For our tests, we used a biased coin – that is, we changed the probability of success from the default of 0.5 to 0.8 (we did this so that more "1"s would be generated). In a few lines of pseudo-code (in the statistical programming language R), this can be summarized as:

```
setSeed(seed);
samples = randomUniform(size);
samples = samples/(10^randomBinomial(size,0.8))
```

Depending upon the probability, this either divided each test case by 10 or did not. We used this method to generate test cases for different sample sizes on the order of 10^7. Note that all test cases were performed with a significance level $\alpha = 0.1$. Figure 9 shows the average runtime for the two methods on the randomly-generated data.

We see a marked decrease in the runtime between the naïve LSU and the FastLSU. Furthermore, as the data size increased, the gap between the two methods increased dramatically. We see the time complexity grow linearly for the FastLSU. The HapMap groups from Stranger's cis-eQTL study [10] are on the order of 1.0×10^7, however, the runtimes for these studies were much smaller that of the timing data from Figure 9 for that data point. We suspect that non-uniform data can produce the fastest results.

We noticed that the rate of growth for the FastLSU is lower than that of the naïve LSU. In particular, if one looks at the rate of increase in runtime from the 1.0×10^8 case and the 2.0×10^8 case in Figure 9, we see a steeper incline for the naïve LSU. We can attribute this to the effect that sorting has on large-scale hypotheses tests. As the data sets grows in order of size, the run time increases in a linearithmic manner.



	1.00E+07	2.00E+07	3.00E+07	4.00E+07	5.00E+07	6.00E+07	7.00E+07	8.00E+07	9.00E+07	1.00E+08	2.00E+08
NaiveLSU (avg)	1509	3032	4454	5795	7151	8980	10062	11581	12602	14592	28610
FastLSU (avg)	651	1219	1763	2614	2975	3505	4155	4715	5250	5463	10953

Figure 9: Runtimes for the two methods. FastLSU grows at a slower rate compared to that of the traditional LSU. Segments = 4. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

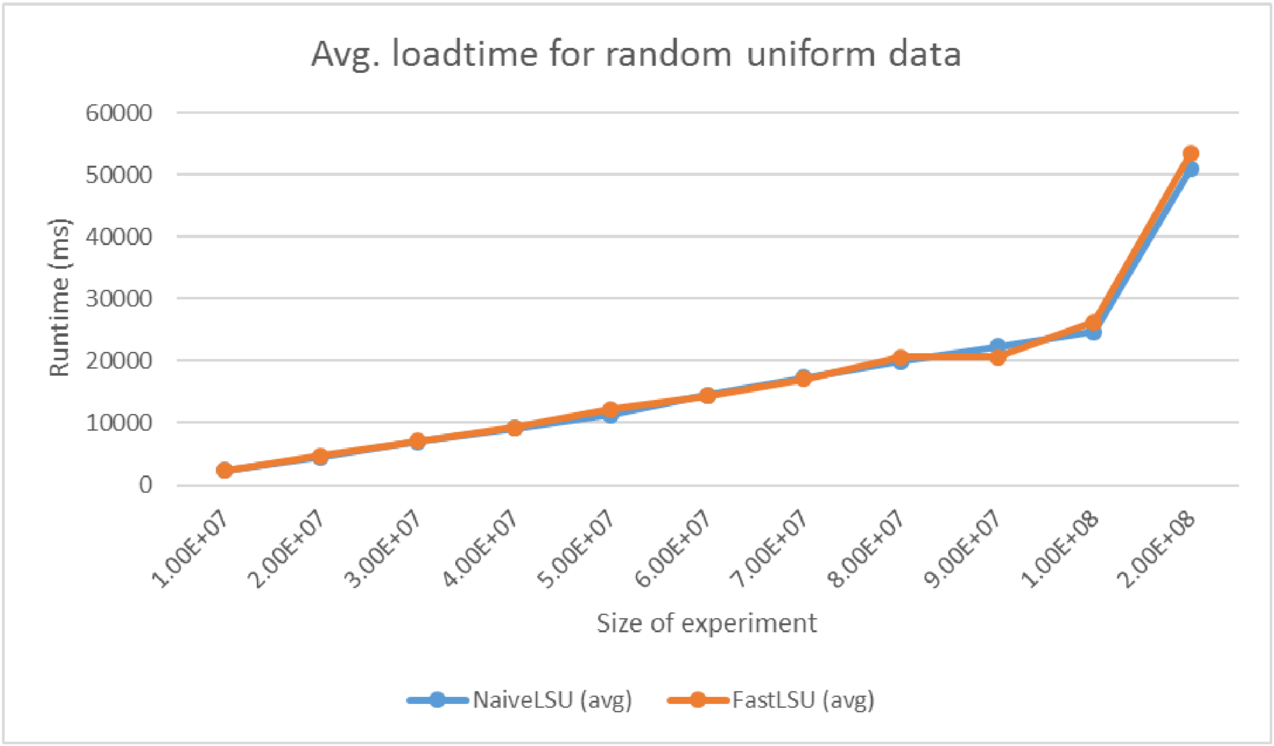
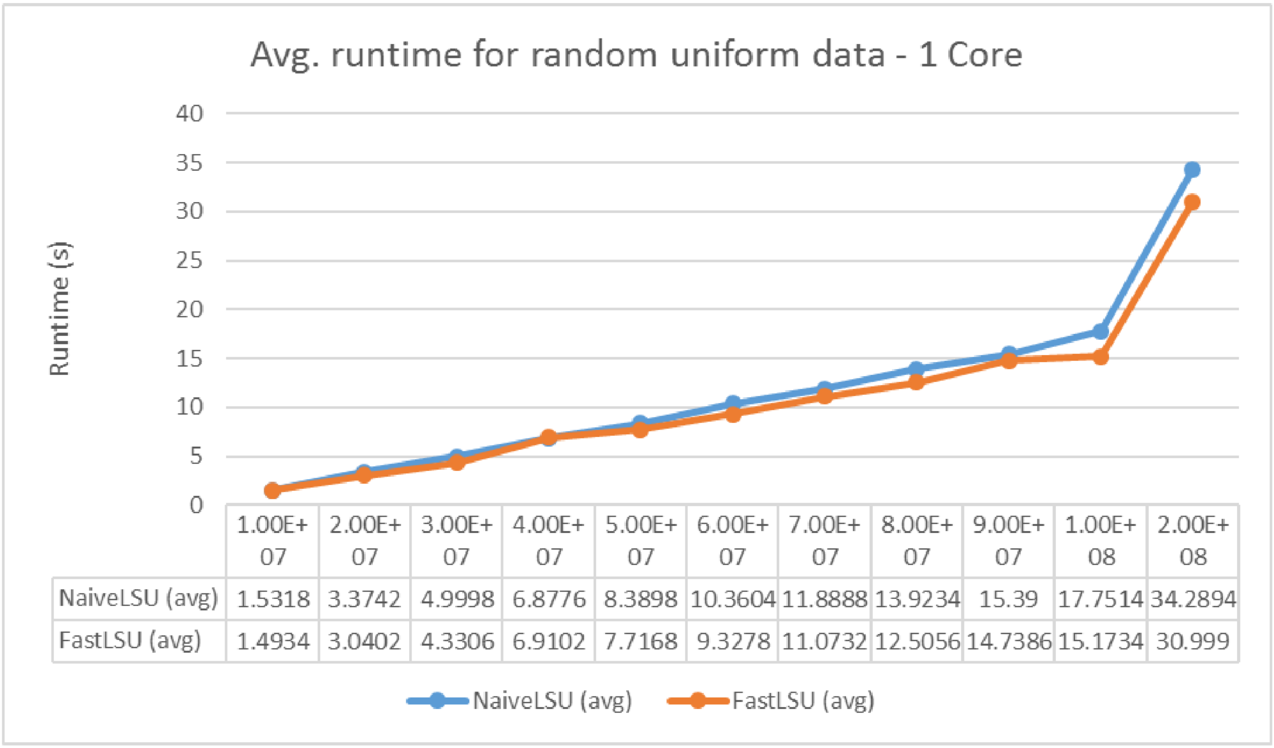


Figure 10: Average load times for the two methods. Segments = 4. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM

Figure 10 shows the nearly-identical load times for the two methods. In our implementation of FastLSU, while loading, we performed the first step of Algorithm 2: we counted all p-values $< \alpha$, and we initialized each core to not only have the subsection of tests, but also the alpha count. Even when doing this, the load times do not differ much between LSU and FastLSU.



	1.00E+07	2.00E+07	3.00E+07	4.00E+07	5.00E+07	6.00E+07	7.00E+07	8.00E+07	9.00E+07	1.00E+08	2.00E+08
NaiveLSU (avg)	1.5318	3.3742	4.9998	6.8776	8.3898	10.3604	11.8888	13.9234	15.39	17.7514	34.2894
FastLSU (avg)	1.4934	3.0402	4.3306	6.9102	7.7168	9.3278	11.0732	12.5056	14.7386	15.1734	30.999

Figure 11: Average run times for the two methods. Segments = 1. Hardware: Intel Core i5-2410m CPU @ 2.30 GHz, 8GB RAM. The scale is in seconds - the linearithmic sort is still slower than the linear algorithm

Since each segment in this tool is handled by a separate thread (one for each core), many of the results take advantage of concurrency. Even in test cases where we forced the tool to only use one core (thus, only one segment – see Figure 11) we still saw a similar difference in the average runtimes across the different sample sizes. We suspect that the reason these times were closer can be attributed to the fact that the data was uniform: in real-world case studies, this may not be so, and as a result in those cases we see remarkable speed-ups.

In this case, since only one segment was used, we performed Algorithm 1 on the data set. Many personal machines are dual/quad-core, yet we still argue that these times are useful for single-core machines - i.e. if the user is using a virtual machine or the tool is on a web server in which the user is offered only one core. In this case, we chose not to segment the data because the advantage of doing so was negligible. If, say, we used segments of size 100,000 when only one core was present, we would have had, in the case of 1.0×10^7, 100 different segments. Thus, we would have had to perform Algorithm 2 *sequentially*, so the wait time for each segment would have been scaled by the number of segments left to process for a given step (the earlier segments would be waiting much longer than the later ones). Instead, we chose to just use a single segment. Even though the big segment itself takes time to process, once it is processed, the step is finished and one can move forward: no waiting required. Furthermore, creating all those segments requires some more memory overhead for each chunk (even though the overhead is small, one major design goal was to limit memory usage in total). Overall, we can justify that FastLSU provides significant performance gains compared to its linearithmic counterpart.

8. More features

8.1. Adjusted p-values

If the researcher desires *adjusted p-values*, or *q-values*, these can be calculated as well after the Fast Linear Step Up procedure has terminated. As outlined in Madar and Batista [7], this can be achieved in a post-processing procedure using the following method:

1. Sort the Fast Linear Step Up results in ascending order. Let 1 to R be the indices of the results.
2. Start from the largest index R. Let P_R be the largest result. Define $Q_R = P_R$ to be the q-value for P_R.
3. Working backwards from the largest q-value (which we just set in the previous step), let $Q_j = \min(P_j \times \frac{m}{j}, Q_{j+1})$, where j is the range of indices from $[R-1,\, 1]$ (j decreases every iteration) and m is the total number of hypotheses.

This method was proposed by Yekutieli and Benjamini [12], and has a linearithmic time complexity (it is linearithmic in terms of R). Since each subsequent q-value is only based on the previous q-value, only one comparison occurs for every p-value. Furthermore, if the user only wants the q-values for a specific subset of the results, then this method would allow that to be done.

9. Future Work

9.1. JIT compilation time

In the future, we would like to see the tool exist on a web server. This would limit the JIT compilation penalty – all the optimizations would occur during server setup. so the runtime would be more dependent upon the user's internet upload speeds than the actual runtime of the procedure. We could then potentially use different languages/web technologies that utilize concurrency and parallelism on multiple machines. Depending upon the frequency of use, if a server had multiple machines and/or many cores, we could also segment the data amongst many different machines, thus taking advantage of concurrency. In order to accomplish this, we would like to use technologies such as Apache Spark and/or Hadoop that are designed for distributed computing.

9.2. Logic changes

Another possible change that could be beneficial would be to, instead of changing values in memory to -1 for failed cases, simply return the condition that resulted in the procedure finishing (this condition would be the value that a given hypothesis was compared against), and then once this value is known, go through the hypotheses and return the values that satisfy this final condition. The reason this could be an improvement is because this would prevent subsequent loads between calls to our tool (i.e. one could just run the procedure instead of having to load it into memory each time). This could save time if, say, a researcher wanted to interactively change the significance level and see the result faster. However, our tool would be utilizing more memory for any given experiment. So, if there were other more data-intensive programs that a user may be utilizing at a given moment, if we were to implement this, then our application could become a resource hog and potentially slow down other applications.

9.3. Performance improvements

In the memory-constrained FastLSU algorithm in Figure 3, we noted that one can load the temporary file into memory and then perform Algorithm 1 on the file and return the results. However, in the case where the temporary file cannot fit into memory, then we perform the following procedure on the temporary file:

1. Load as much of the temporary file into memory as possible. This is segment 1. Then treat segment 2 as the remaining tests in the temporary file.
2. Perform algorithm 2 on both of these segments, with the initial condition for both segments being $\alpha \times \frac{m'}{m}$, where m' is the size of the temporary file. After this, the temporary file is deleted.

In the current implementation, segment 2 is processed by multiple I/O calls (opening the file, checking the p-values that pass the condition, closing the file, and then repeating until algorithm 2 is finished). Unfortunately, if the temporary file is significantly larger than the amount of tests that can fit in memory, then the repeated I/O calls will significantly impact the overall runtime. Currently, the way to mitigate this problem is to a) use as much memory as possible under the defined memory constraints (Section 3.1 or b) use a machine with more memory. Besides methods to decrease I/O times (using a Solid-State Drive as opposed to hard disks), another possible solution to this problem would be to perform the algorithm in a distributed cluster (this way memory becomes a non-issue for large-scale hypotheses tests). Still, for individual machines, in the future we would like to develop a solution that would limit the number of I/O calls for this particular case.

9.4. Room for improvement

Other general areas of improvement include tweaking the user interface over time so that it better serves the end user. In addition, we could also consider loading the entire data set (if it fits) into memory and keeping it there). This would speed up operations for instances where, say, the researcher slightly tweaks the significance level and wants to see the difference between results. This could be an option, though overall the tool could be hogging a significant amount of memory at any given time.

9.5. Challenges

In terms of the logic of our implementation, we would like to explore different segmentation methods to see which ones perform the best. With this tool, we assumed that segmentation works best when the data is split amongst the number of available cores. However, since the actual procedure works on an arbitrary number of segments, one area of research could be to find an optimal segmentation procedure. We stipulate that if a more optimal segmentation method exists, then this procedure would have to be parallelizable. If one were to arbitrarily segment an experiment into different chunks and processed these chunks sequentially, there would be many of the previous chunks all waiting for the subsequent chunks to find their counts before the earlier ones can do any comparisons. In ou current implementation, since the segments are being processed concurrently, the wait time for any given segment is low, assuming the segments are approximately equal in size. If one were to segment and process *sequentially*, then each segment's wait time would be scaled by the number of remaining segments that need to be processed. Thus, a potential optimal chunking method would indeed utilize concurrency.

10. Conclusion

Our main goal was to provide researchers with an easy-to-use application that allowed the user to control the false discovery rate in an experiment more efficiently than in the past. We sought to first implement the algorithm by utilizing core Java datatypes as well as some of our own. After this, we then created a very simple GUI using the JavaFX package to make it easier for those not well versed in computer science to use. By comparing the runtime for the newer FastLSU procedure versus the original LSU paper, we show that the runtime of the FastLSU is significantly lower than that of the original algorithm. We recommend that for experiments on the order of 10^6 or lower one can simply use the naive LSU method. The performance gains of FastLSU are not as easily seen on such a small scale.

In addition, the runtime grew linearly as the sample size increased. Furthermore, load times were nearly identical in both cases. We anticipate that the reduction in processing time along with the ease-of-use will entice users to utilize this tool over other statistical programs such as R or Stata. While these statistical packages are no doubt beneficial for statisticians and bioinformaticists, in some cases ease-of-use is sacrificed for more features. We aimed to keep the number of features we implemented to a minimum. Nevertheless, the applications for this kind of tool vary. We foresee researchers using this tool for neurological research, genome wide association studies, microarray analysis, astrophysics, brain imaging research, and other computational biology fields. The statistical procedure that we implemented can also be used as a backend method for a variety of statistical programs such as R or Stata or other software in the field of computational biology.

We were also able to add an extra feature to generate the q-values for a recently-performed experiment or older experiments from the user's file system. The logic of our design lends itself towards future improvements, such as creating an implementation optimized for distributed computing (if the experiment is on an enormous order of magnitude in terms of number of tests). Ultimately, this project is an open-source project, so we hope to continue to refine and tweak our tool and/or have individuals in the future contribute to the project in order to better serve the researchers that will be using the tool for their experiments.

References

[1] [Online]. Available: https://attractivechaos.github.io/plb/

[2] *JavaFX*. [Online]. Available: https://docs.oracle.com/javafx/

[3] *VisualVM - Java*. [Online]. Available: https://visualvm.java.net/

[4] *How people build software - GitHub*, 2016. [Online]. Available: https://github.com/

[5] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological), Vol. 57, No. 1*, vol. 57, no. 1, pp. 289–300, 1995. Available: http://www.jstor.org/stable/2346101

[6] G. Kowal, *Cross-platform Java executable wrapper*. Available: http://launch4j.sourceforge.net/

[7] V. Madar and S. Batista, "FastLSU: a more practical approach for the benjamini–hochberg FDR controlling procedure for huge-scale testing problems," *Bioinformatics*, pp. 1–8, jan 2016. Available: http://dx.doi.org/10.1093/bioinformatics/btw029

[8] T. V. Perneger, "What's wrong with bonferroni adjustments," *BMJ: British Medical Journal*, vol. 316, no. 7139, pp. 1236–1238, April 1998. Available: http://dx.doi.org/10.1136/bmj.316.7139.1236

[9] J. D. Storey, "The positive false discovery rate: a bayesian interpretation and the q-value," *The Annals of Statistics*, vol. 31, no. 6, pp. 2013–2035, December 2003. Available: http://dx.doi.org/10.1214/aos/1074290335

[10] B. E. Stranger *et al.*, "Population genomics of human gene expression," *Nature Genetics*, vol. 39, no. 10, pp. 1217–1224, September 2007. Available: http://dx.doi.org/10.1038/ng2142

[11] E. W. Weisstein. Bonferroni correction. Available: http://mathworld.wolfram.com/BonferroniCorrection.html

[12] D. Yekutieli and Y. Benjamini, "Resampling-based false discovery rate controlling multiple test procedures for correlated test statistics," *Journal of Statistical Planning and Inference*, vol. 82, no. 1-2, pp. 171–196, December 1999. Available: http://dx.doi.org/10.1016/S0378-3758(99)00041-5