Estimating N with Duplicate Confirmation Sampling Summary Report

Initial Problem & Background:

Let $S = \{X_1, X_2, ..., X_n\}$, where n is unknown. The only function and interaction with the given set allowed is sampling with replacement. A random element is chosen from the set, looked at, and placed back in the black box. The black box is shaken, and the sampling can repeat an infinite number of times, if needed. Can an algorithm be designed to estimate the actual value of n within a certain margin of error based on sampling with replacement? Over 6 weeks, starting from May 1st, 2019, many prototypes of algorithms were implemented, tested, and corrected based on a combination of previously known algorithms related to general sorting, medians, sampling, population estimates, and Prof. Smid's proofs. This is a report providing an overview of what was attempted, what failed, what succeeded, and what was learned at each step until the final algorithm for estimating n (called EstimateN) was achieved. The code files corresponding to each portion of the report are available for reference in the NSERC-USRA-2019 Github repository and referenced accordingly. The files referenced in the last section are found in the EstimateN directory in the Inactive_Code folder of the repository. All others referenced are located in the EstimateN_Experimental_Prototypes directory in the same folder.

Medians:

It was initially thought that the median would be a value of interest, around which perhaps the rank of the value within the entire set could be established, so different algorithms for determining the median became the starting point. One algorithm sampled the set 2n+1 times, sorted the samples, and then took the median. Another method of took the median of medians when looking at groups of 3 (or 5) samples. These algorithms were presented, and the proofs behind each were explored, allowing for the introduction of Chernoff Bounds, Chebyshev's Inequality, and Markov's Inequality. It was then quickly discovered that the median of a set had no relation to the spread of the set, and could not give any useful information, so these algorithms were set aside, though their basic methods for sorting to approximate a correct value were kept in mind moving forward.

Initial Algorithm:

Another concept of interest was confirmation sampling, in which a set is sampled until any given element was seen an arbitrary threshold number of times. This sampling could reliably determine the min/max and k-th smallest/largest elements, depending on the number of times it was run. This kind of sampling, which was the fundamental building block of the algorithm can be found in the Sampling_Until_Duplicate.py file of the EstimateN directory. Then the Birthday Paradox was introduced as an upper bound for sampling. According to the Birthday Paradox, any given set S of size n would only have to be sampled sqrt(n) times on average before a duplicate appeared. These two concepts were combined to form duplicate confirmation sampling, in which element sampling happened at random until a duplicate was encountered. The expected value of sqrt(n) became of interest as it could be easily obtained and was related to the overall size of the set. An initial algorithm based on the Birthday Paradox came about in which it sampled until duplicate, squared the number of samples it took to get that duplicate, and multiplied it by some constant. A proof done using set theory probability proved this constant to be:

$$C = \frac{1}{2ln(2)}$$

The idea of also somehow determining the rank of the median based on this approximation of the set size, and then using that to further hone the estimation, was proposed around this time, although no implementation was set down.

Proof & Verification of Algorithm Parts:

There was a series of tests to iron out the specifics of different parts of the algorithm that used this constant C. The expected value for the number of samples to take compared to a practical run of the algorithm was compared in the Comparing_Counter_To_Expected_Value.py file, and it was verified that on average, the implementation supported the theoretical values. Then the accuracy of determining the actual value of the median of a set by using either the median of medians, or the median of averages was tested through the Median_Of_Medians_Vs_Median_Of_Averages.py file. It was discovered that taking the median of averages tended to be more accurate, as the effect of getting skewed samples was mitigated by averaging all the results. Then Prof. Smid presented a proof that the margin of error, Pr[samples not in green], would be upper bounded by a negative exponential function related to the number of samples and the delta chosen, due to Chernoff bounds and a magical assumption of being able to somehow know the rank of the median. This figure was tested to hold true in practice in the Not_In_Green_Interval_Error_Bound_Inequality.py file.

German Tank Problem:

After searching through statistical discipline-related methods and finding them lacking in the desired precision, the German Tank Problem was discovered. In the original method, the set size was calculated based on the sample maximum and the sample size using a sampling of only a portion of the total set without replacement. A modified version that used the entirety of the set and sampled with replacement was tested out in the Modified_German_Tank_Problem.py file. Multiple versions were tested, including using different methods to obtain sample counters. It was established that it would not work accurately for the purposes of this problem due to its low accuracy rate, hovering around only 40%. Then it was discovered to be a looping one-off issue that made the estimation so far off, but even after correction, the accuracy could not satisfy the deltas given for a small epsilon.

Multiple Algorithm Implementations:

Then a brief attempt at using a counter from taking the median of 2k+1 counters to approximate the set size was implemented in the Approximating_N_With_2kPlus1_Samples.py file. This was due to a misunderstanding and mix-up in terminology on Julia's part. Multiple attempts at tweaking the overall algorithm happened in quick succession. There was a version to estimate n using the median of the counters, implemented in Approximating_N_With_Median_Counter.py. This was routinely off due its tendency to rely on the probability of getting enough good samples to have good median to use. Then Prof. Smid introduced the idea of the Law of Large Numbers to use averages for determining a good sample counter to use in the final calculation. This was then implemented in the Approximating_N_With_Average_Counter.py file, and tested appropriately in the Approximating_N_With_Average_Counter_Test.py file. Now that there were two algorithms, differing only by their counter used, the algorithms were compared to see which was better in the

Median_Counter_Vs_Average_Counter.py file. It turned out that using the average was better overall due to the Law of Large Numbers allowing for the value to eventually converge to the actual expected value of the sample counter. At some point, Luis proposed his own version to implement, which can be seen in the May_14th_Luis_Homework.py file, although that attempt was not addressed after its proposal. Then, it was discovered that the constant C from above was only applicable to the version of the algorithm using the median counter, and that a different constant C was obtained when doing the proof for the efficacy of the version using the average counter. This new constant was found to be the following:

$$C = \frac{2}{\pi}$$

After using this in the implementation of the average counter version, the accuracy went up and the version using the median was firmly pushed aside.

K-Matrix & Attempts at Multi-Processing/Threading/Cores:

During this time, many overnight tests were done each implementation and attempt at a new algorithm or a tweaked version, but these were slow programs that took minutes for a single run. The idea of making performance gains through learning and implementing a version of multiprocessing into the code arose as the need to discover the relation between epsilon, delta, and the number of samples in order to determine the optimal number became apparent. To determine this best k value relation, it was necessary to create a matrix of experimental to results from which to draw conclusion, for which more speed was needed, or else testing would never be completed. There are a slew of files relating to this. The files Multiprocessing_Practice.py, Multithreading_Practice01.py, and Gradual_Buildup_Multiprocessing.py all illustrate learning the Process and Thread classes of the multiprocessing and multithreading modules in Python starting from basics to initiating call-backs and pools of workers. The file Multiprocess_Number_Doubler.py is an attempt to max out the CPU to test the limits of Python on a machine. The ThreadPool_Practice.py file experimented with the Pool class of the module. Then an attempt at improving performance on the overall algorithm itself can be seen in the Multiprocess_Approximate_N.py and Multiprocess_Approximating_N_Test.py files. Ultimately, the multi-processing attempts were a failure as it was discovered that the Python language itself put hard limits on such capabilities. The Global Interpreter Lock (GIL) allowed only one task to be completed at a time and blocked off access and resources from all others, irrespective of the creation of a pool of workers or multiple threads and the availability of multiple CPUs. There was even an attempt to circumvent this through Linux system commands in the Subprocess_Practice.py file, but eventually it became clear that the language itself was the issue and the decision was made to port the code over to C++ for performance. Meanwhile, it was also discovered that at very small epsilon-deltas and set sizes, the algorithm would consistently overshoot in its estimation. It was discovered through a very thorough proof by Prof. Smid that there was a series of extra constants and trailing figures after the Big O value which held heavy sway over results at extreme values in the algorithm. Therefore, it was deemed necessary to include the subtraction of 2/3 in the final calculation of the set size.

Binary Search & Testing:

During this time, and including the events of the next section, the overnight tests slowly revealed the exact relationship between epsilon, delta, and k. To search for a correct number of samples to take, binary search was implemented in the form of the Binary_Search_Best_K.py file. Each of the results were collated into spreadsheets, with the original raw output residing in the

Original_Test_Results_Compilation.ods file, and the analysis of the results going into the Best_K_Test_Results.ods file. It was discovered, with proofs suggesting it and the practical evidence backing it, that the main value of importance was epsilon. To save on time, a proof was done to showcase how epsilon would square-inversely relate to the value k if delta was fixed and this was integrated into the test code. The final results showed that the actual relationship constant between epsilon and k was 1.2632. This can be seen in the final EstimateN.py file that integrates all portion of the algorithm and learned knowledge into a program that can correctly given an approximate of n within a given margin and probability of success.

Final Version & C++:

For the greatest accuracy according to a proof that include Chernoff Bounds, another modification to the algorithm had to be made. The number of samples to take would be dependent on the best k valueTo port the code from Python to C++, the expertise of a colleague in the lab, Luis Fernando Schultz Xavier da Silveira (PhD) was enlisted, and through his patient guidance, the two C++ versions of the code were created. The files in the N1 directory contain the original ported version of the algorithm, with the main program residing in the N1.cc file. Since many new tools were introduced in the porting process, there is a heavily commented version for educational and learning purposes named N1_Commented.cc that dissects all parts and choices of the program, including the Makefile. This version of the algorithm ended up being a whopping 125 times faster than the Python version, which was possible due to aggressive optimization choices and specific lines added for time shortcuts, particularly for large set sizes. For further performance improvements, a multi-threaded version was also created. The N2 directory contains all of these files, with the main program residing in the N2.cc file. This version runs 250 times faster than its Python original, which just showed the dominance of C-level languages in performance optimization and the benefits of using languages closed to the hardware as opposed to high-level scripting languages in testing. For learning, a commented version of N2.cc was created as well, the N2_Commented.cc file which equally scrutinized all the design decisions and tool usages in the program and the corresponding Makefile. Thus, the problem of creating an algorithm that could approximate set size through duplicate confirmation sampling with replacement alone was solved and the results of it are fast C++ programs in the EstimateN directory.

***Extension* - Coupon Collector's Problem:**

A slight extension on the problem of approximating set size was brought up as a modified version of the Coupon Collector's Problem. The conventional Coupon Collector's Problem asks how many times one has to sample before every element in the set has been seen. The unmovable lower bound for this is as follows:

$$E[\#samples] = n * Hn + n$$

with Hn being the n-th harmonic number. Based on this known lower bound, a version of this problem was combined with the above algorithm for estimating set size. Due to the interval of acceptable values for n above, the number of samples to take was calculated based on the approximated n, and the value of epsilon as follows:

$$Sample\ m\sim * \ln(m\sim) + C * m\sim\ elements$$

A program was implemented in C++ to determine the best value for the constant C such that all elements were seen. This program sampling estimated the set size, took the required number of

samples, and checked at the end whether every element had been seen or not. This algorithm exists in its original single run form in the C1 directory, with C1.cc being the main program. It utilized all the methods used in the N-series files for performance optimizations, with some slight modifications to existing code. The test version, with the Coupon Collecting Problem separated out as its own function was relocated to the C2 directory, with C2.cc being the main program. After running tests, it was discovered that the constant C was essentially useless at large enough set sizes due to the margin of error in estimation always guaranteeing that enough samples would be taken to see all elements, but at smaller set sizes (ie. < 100 million), there was a positive correlation between C and the probability of success. C itself did not need to be large, as it was only the coefficient related to the set size, but as C went towards 4, the chance of success went to 1. The larger the set size, the quicker 1 was achieved. It was surprising however, to discover that delta had only the most negligible effect on the probability of success. As delta went to 0, the success rate was negatively impacted, suggesting a positive correlation, but the variance in results was not significant, only differing by a few percentage points at most amongst thousands of runs. Epsilon had the largest impact in that it was exponentially related to how many samples were taken due to the calculation of m~ from n~. The larger epsilon became, the more explosive the number of samples to take grew, thus a positive exponential correlation was discovered between success and epsilon. The relations for the constant C and epsilon made sense intuitively, but tests showing the lack of effect from delta was a surprise to be sure.