An Algorithm for Memory-Efficient Solutions to Large Graph MST Problems

Arjun Bhalla, *ab2383@cornell.edu*
Ali Abouelatta, *aa2767@cornell.edu*

Cornell University: Cornell Tech
Dept. of Computing & Information Science
CS 5112: Algorithms and Data Structures for Applications

1 Introduction

Finding the minimum spanning tree (MST) of a graph, i.e. the set of edges which connects the entire vertex set with minimal cost, is a popular and well studied category of graph problems [1]. There are a number of algorithms that have been used to solve such problems, notably Kruskal's, Boruvka's, and Prim's Algorithm.

Minimum spanning trees have direct applications in the design of networks (including computer networks, telecommunications networks, and transportation networks), clustering gene expression data [3], image registration [4], and image segmentation [5]. Often, these tasks will be quite memory intensive, especially as graph sizes scale, and thus can limit performance or even the problem feasibility. This is especially true in a memory constrained environment (e.g. GPU systems utilising shared memory) or if processing massive graphs that may not fit in memory (resulting in potential thrashing issues) [6].

To solve this problem and reduce the memory requirement needed for finding an MST, we propose a novel approach using Bloom Filters. Bloom Filters are a space-efficient randomized data structure for representing a set in order to support membership queries, which allow false positives (importantly, not false negatives), but the space savings often outweigh this drawback when the probability of an error is controlled [2]. In this paper, we modify the optimal implementation of Prim's algorithm with Bloom Filters to investigate the difference in memory burden from the original hashset-based implementation. We compare the memory requirement and error rate of introducing a Bloom Filter in lieu of the hashset to track visited nodes, hypothesising that this will lead to a notable decrease in memory consumption, with some acceptable error bound for most applications.

2 Algorithms & Data Investigated

2.1 Algorithms and Baseline Research

We began our research looking at how to solve general graph problems with greater memory efficiency. Thus, we began by looking at Dijkstra, Bellman-Ford, Ford-Fulkerson, various MST algorithms (elaborated on below), pathfinding algorithms (BFS, DFS, A Star, ID-DFS, etc.), and other assorted graph algorithms.

However, we decided to focus on MST problems for two main reasons. The first was their widespread usage and importance in both indirect, real-world applications (i.e. logistics and networks), as well as directly within Computer Science specific tasks (e.g. Image Registration, Clustering, etc.) [5]. The second was that we saw room for improvement, optimization, and expansion of these methods that was available to us using the tools at our disposal. Upon realising that there may be a more space-efficient way to solve these problems using the knowledge that we gained from lectures (later settling specifically on Bloom Filters), we did more in-depth research on the following algorithms - specifically, their implementations, special use cases, runtimes, and space efficiency:

- Prim's Algorithm: $O(|V|)$ space, $O(|E|log(|V|))$ time (Binary Heap, PQ), $O(|E| + |V|log|V|)$ time (Fibonnaci Heap, PQ) [7]
- Kruskal's Algorithm: $O(|E| + |V|)$ space, $O(|E|log(|V|))$ time [8]
- Boruvka's Algorithm: $O(|V|)$ space, $O(|E|log(|V|))$ time [9]
- Bloom Filters: Set membership check, entry: $O(k)$ time (k is the number of hash functions used, a parameter of the Bloom Filter)
- Locality Sensitive Hashing

Eventually, we settled on Prim's algorithm, because it is one of the most popular MST solving methods [1], and it is comparable to and occasionally even asymptotically faster than the other popular MST methods.

2.2 Data Generation and Usage

We chose not to use a dataset from a potential application area, for example Image Segmentation or Gene clustering, because the purpose of this paper is to be exploring a new approach to solving a general problem, we wanted to therefore test across the most general set of data possible. As a result, we decided to write an algorithm that would generate a random graph based on controlled parameters (one giant component). This allowed us to control data size (not having to artificially trim or duplicate datasets), noise, sparsity / density (we had control over the min and max number of neighbours of a given vertex), and any other features we may have needed to fine tune to test any of our potential hypotheses (more on the generation code below).

We strongly believe that this was a better approach than choosing a dataset from elsewhere (which we also considered), because of a) the nature of our problem & objective requiring generality of data, and b) the fact that we would need to build different custom implementations of the algorithm to work cleanly with every dataset, so having a standard, consistent source of data that wasn't prone to strange or inconsistent formatting would be of paramount importance - from experience, this can be relatively challenging.

3 Technical Notes

3.1 Runtime & Asymptotic Complexity

It is important to note before we begin that there is a downside to augmenting this algorithm aside from the accuracy considerations - while the asymptotic running time of the baseline implementation is $O(|E|log|V|)$, our Algorithm's runtime is $O(k|E|log|V|)$. This is because we need to query the Bloom Filter each time we pop from or add to the heap, and each operation requires computing k hashes. In one iteration, these scale linearly with $|E|$ because we will only consider each edge once.

Practically speaking, however, this is almost negligible for the size of dataset that we are handling, because empirically k is quite low (usually ~ 6).

3.2 Calculating Bloom Filter Parameters

There was much debate on how to calculate the Bloom Filter parameters - should we run experiments varying one parameter (hash functions, table size) while holding the rest constant? Or should we just maintain one set of parameters and vary input size? We eventually decided to use accepted calculations of the optimal values of k and m, given below, for a number of reasons. First, the purpose of this project was meant to be an experimental foray into this unknown approach, so spending time manually conducting hyperparameter tuning could have been distracting and potentially not give us much insight for the limited timeframe on which we were operating. Second, these formulae allowed us to use the classic Software Engineering principle of abstracting implementation details away from the user, which gave us enough flexibility to set up future work and easily apply this to a real example, while still maintaining control of the overall important features; This was done by allowing the user to adjust desired error rate and graph input size, but not confusing them with Bloom Filter implementation specifics. Finally, this approach allowed us to write fairly bug free code - by having a single calculation and minimal input values, it was much easier to debug if and when the Bloom Filter was having issues, as opposed to if we had "magic numbers" and different variables floating around. Of course, if part of our future work was to experiment with these parameters, that could easily be facilitated too.

The formula for calculating the optimal number of hash functions is $k = \frac{m}{n} ln2$, and the formula for calculating the optimal table size is $m = \frac{-nln\epsilon}{ln(2)^2}$ [16]. Here, k is the number of hash functions used in the filter, m is the number of bits in the table, n is the size of the data, and ϵ is the desired error rate.

4 Code Written

We wrote a fair amount of code for our assignment, since the project was implementation based and required a lot of algorithm tweaking, graph generation, plot generation, analysis, and re-running to test hypotheses. Thus, there is too much to just paste here, but the full code with good documentation is available at a public Github Repository[1]. However, the overall code structure and implementation philosophy are laid out below.

Generally, one experiment (1,000-101,000 node graphs in 10,000 node increments, generating the graph, running both algorithms, plotting) took about 2 hours to run. This was mostly graph generation cost.

[1] `https://www.github.com/ArjunBhalla98/bloom-filter-prims`

4.1 Graph Generation

The first stage of the project was to build the ability to generate graphs in the first place. This was quite straightforward - we wrote our code to allow the user to input a size, max neighbours (min neighbours was defaulted to 1: we wanted a giant component so we couldn't have any solo nodes). This would generate n unique "nodes", which we would then iterate through and randomly choose neighbours for. Later on, when we moved past just estimating the cost and added on the full-graph approximation, we added a variable that would assign each edge a unique "index" to be referenced later (See 4.3).

4.2 Bloom Filter

This was quite a straightforward implementation - we built a class that took in an acceptable error value (epsilon) and a dataset size, and from that, used accepted formulae to calculate the optimal value of k (number of hash functions) and m (size of Bloom Filter bit array), which are discussed in more detail later. An interesting detail here is that we were getting somewhat strange and nearly suboptimal results when we used a simple array implementation for the Bloom Filter (i.e. boolean array). Because of Python's dynamic typing and thus lack of memory optimisations here, the boolean array had entries which were 8 bytes each + overhead (discovered through manual testing). This was an obvious place for improvement, and through using the python `bitarray` package [10], we were able to reduce this memory consumption by essentially an order of magnitude.

An important implementation detail here was the choice of library that we used: we used `murmurhash3`, a python3 library that provides variable output length universal hash function families, which would return a completely different (but of course, deterministically different) universal hash function given a seed [11]. This made determining the buckets in which to check/flip the bit for a given element fairly straightforward, as we would simply loop through the range of k and use each loop iterator value as a different seed.

4.3 Algorithms

This was quite straightforward, there is not much nuance to the implementation details or the philosophy here. We implemented the basic, well known implementation of Prim's algorithm with the Priority Queue from pseudocode [7]. We found later however that the Python implementation of the native `heapq` library is actually a binary and not fibonacci heap, so we acheived the slightly less than optimal implementation time-wise. However, we feel this does not affect our findings because the Bloom Filter focuses on the use of the hashset and our implementation is agnostic to the underlying workings of the heap.

To get the Bloom Filter adaptation of this algorithm, we simply copied the baseline to another method in which we adapted the implementation by initialising the Bloom Filter upon seeing the data, and replacing any set query or addition with a call to the corresponding Bloom Filter method. We also had a counter on each which kept track of the total cost of the tree for each method.

We also added a bitarray the size of the number of edges to each method, which tracked which edges were selected for the final MST in each case (as mentioned earlier, we mapped a unique edge index to each edge. We thought this would be a reasonable approach as in a regular application, it wouldn't be tough to do initially or during the pre-processing of the graph). This allows us to not only fully represent the final MST, but also compare the results of the two methods directly.

4.4 Plot Generation

There is not too much to say here, except that we deliberately chose to build out a full class with abstraction and good engineering practices because we knew that we would want to run many different tests with graphs with varying parameters, so to aid in this, clean, abstracted, and efficient code was required, especially because plotting a graph is quite a few lines of tedium. This came in handy when we wanted to run some quick experiments and sanity check some new theories or directions, because the univariate and multivariate plot functions were already abstracted out. Overall, the effort was certainly worth it.

We essentially wrote two classes: one that would be able to run metrics effectively, customize input size and other internal variables (i.e. sparse v dense) that could be used as a backend to our "Plotter" class. This class was the interface through which we automatically ran tests and plot the requisite graphs in the correct folder, with dynamic and premeditated naming conventions. The next step, if we needed to take it further, would be to add in command line arguments when calling the plot script (`run_metrics.py` in the github) instead of manually changing variables in the code itself.

5 Experimentation

5.1 Metrics and Methodology

To measure the impact of our modified Prim's algorithm against the baseline and validate our hypotheses, we tracked two main metrics:

- Memory Consumption: This was the primary and obvious metric, as it allows us to validate and quantify our hypothesis that we could solve the problem with a reduction in memory usage.
- Error Rate (in Bloom Filter implementation): This is the secondary metric that allows us to actually quantify how accurate our solution is to then determine whether it is at an acceptable degree.

Error rate was defined in a few different ways - originally, it was just the difference between the calculated cost of each MST. However, this was not a good metric as the relative magnitude of edges have no bearing on the actual MST, so having an incorrect edge could show a cost difference of 0.01 or 1000. We then counted the edge set difference for both, and compared this with the total number of edges in the graph. This gave us more insight into the true number of errors and was indifferent to the nuances of specific edge weight sets, but it was skewed by the edge density of a given graph. Thus, our final error rate metric compares the delta in the edge sets of the two MSTs against the total edge weights in the baseline implementation's MST.

For memory consumption we measured the MST representation storage space consumed by both of our algorithms: for the regular variant, this was the size of the visited set, and for the Bloom Filter variant, this was the size of the edge-mapped bitarray and the Bloom Filter itself. We have deliberately chosen not to graph the running time, although we kept a note of how it differs between both algorithms (which is not noticeably, at least at this dataset size). We did this because it wouldn't have much meaning - we wrote the algorithms in Python, and simple implementation details could cause speeds to vary wildly.

As explained earlier, we used our own graph generation algorithm to create a generalized data-set to test our algorithm on. The experiments we ran involved running both algorithms on different graphs ranging from 1,000 to 101,000 nodes in increments of 10,000 and measuring the memory consumption and error rate on the baseline and modified Prim's algorithm, plotting a graph following each experimental run. Importantly, so as not to bias our findings with a lucky graph, we would randomly re-generate the graph

at every experimental run.

5.2 Results

Results of running experiments on graph sizes from 1,000 to 101,000 nodes in 10,000 node increments.

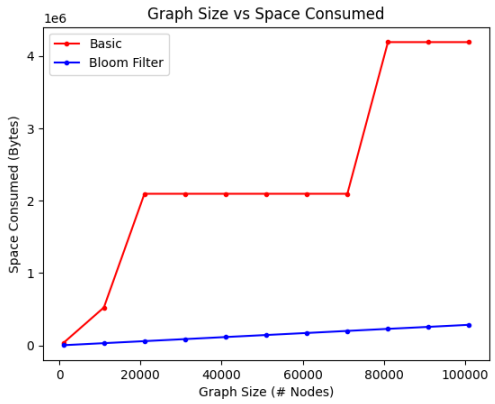

Figure 1: Memory Consumption

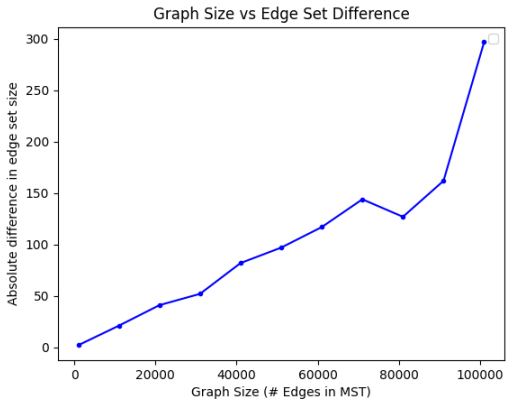

Figure 2: Error

# Nodes	Baseline Space (Bytes)	BF Space (Bytes)	Difference	# BF Incorrect Edges	Error
1,000	32,984	2,944	91.07%	2	0.20%
11,000	524,504	31,304	94.03%	21	0.19%
21,000	2,097,368	59,538	97.16%	43	0.20%
31,000	2,097,368	87,632	95.81%	51	0.16%
41,000	2,097,368	116,058	94.47%	85	0.21%
51,000	2,097,368	144,064	93.13%	94	0.18%
61,000	2,097,368	172,294	91.79%	119	0.20%
71,000	2,097,368	200,775	90.43%	147	0.21%
81,000	4,194,520	229,231	94.53%	127	0.16%
91,000	4,194,520	256,539	93.88%	159	0.17%
101,000	4,194,520	285,219	93.20%	295	0.29%
Average	N/A	N/A	**93.59%**	N/A	**0.197%**

Figure 3: Table of results from experiments

5.2.1 Bottom Line: Key Takeaways

- Achieved an average **93.59%** decrease in memory usage for visited set computation with our method.
- This was with a **0.197%** average error rate - that is, the Bloom Filter implementation missed about 1/500 edges in each graph.
- An interesting phenomena that did not immediately occur to us was that this algorithm will never add an incorrect edge to the tree - the sole source of its error is from missing edges. This occurs

because the Bloom Filter records the nodes that have been seen, and will never change once it has "seen" a node, which means that the any time an error occurs (false positive on a node), the algorithm will omit any connection to that node on the graph, because it believes that node has already been visited. We discovered this when we realised that the cost was always strictly lower or equal in the Bloom Filter implementation, but never higher.

5.2.2 Memory Consumption: Explanations & Inferences

As we can see from Figure 1 and Figure 3, there is a stark difference in the memory consumed by both of the methods, more so than we were hypothesising - our implementation is almost two orders of magnitudes more memory efficient. The Bloom Filter graph follows a shallow but constant linear trend, which makes sense as we can see by the formula given in Section 3 that the size of the Bloom Filter scales linearly with the number of nodes in the graph, and the size of the edge-mapped bitarray scales linearly with the number of edges.

The rather erratic graph for the baseline implementation in figure 1 can be explained simply by the way in which hashsets work - the $O(1)$ lookup / addition is amortized because when a certain number of elements are stored in the set, to prevent excess chaining and thus longer lookup / addition times, the set dynamically increases its size to meet new demand. That appears to be what is happening here, and would explain the jumps perfectly.

We can thus infer that there is what appears to be a shallow but strong linear relationship between graph size and space consumed by our method. This suggests that these results directly support our hypothesis and are unlikely to be caused by some error or unaccounted for factor.

5.2.3 Error Rate: Explanations & Inferences

Again, this seems to be fairly linear with the size of the graph as we see in Figure 2 (since we know that an MST should have n-1 edges at most by logic), which is consistent with the mathematics in Section 3, but the somewhat erratic nature of some data points serves as a further reminder that the error percentage that was calculated is not a guarantee, but rather an empirical finding that supports a linear trend. At the end of the day, Bloom Filters are based on randomness and thus this must be accounted for when considering applying them.

What we can infer from this is that there is clearly some form of a linear trend and relationship between the two variables here, as Figure 2 and Figure 3 show that the error rate seems to grow linearly with the graph nodes size, and thus perhaps in the future we can derive some sort of bound or a more meaningful relationship outside of simple linearity.

Moreover, this and the insight from our key takeaways suggests that due to this linear relationship, error scales with the number of nodes in the graph as opposed to the number of edges, which means that this method should perform equally well on both sparse and dense graphs.

6 Conclusion

6.1 Conclusion from Results and Key Takeaways

We have introduced a new method for solving MST problems that uses orders of magnitude less memory than Prim's algorithm, a traditional and very popular baseline, while introducing a negligible slowdown and what appears to be an acceptable error rate. This will always result in a cost less than or equal to the true cost (provided all edge weights are non-negative). The key takeaway is that this method seems to be better than Prim's algorithm when dealing with large graph problems in memory constrained situations (e.g. GPU graph computations [6]), and where producing an approximate result is acceptable.

The direct implications of this are, assuming that this level of error is acceptable (it can be reduced at the cost of extra space), solving MST problems on massive graphs will be much more accessible commercially, not requiring large amounts of RAM or specialised hardware, as well as providing another approach for constrained memory GPU based graph processing problems [6]. This could, depending on the results of future work, mean that another avenue of approaching problems in application areas such as fast image segmentation or image registration on massive datasets would be available to students and working professionals with hardware constraints, or that GPUs would be able to efficiently solve large MST problems. For example, a dataset that would take 100GB of memory to process using Prim's algorithm could be approximately solved with only $\sim$ 6.47GB using this method.

6.2 Future Work

We believe that this was a promising beginning to what will hopefully be a fruitful and useful application of Bloom Filters. However, there is still much that is left unexplored. We hope to soon apply this to an application area, with a real dataset, and see how it performs against some standard algorithms. We would have to choose a problem space where we would not only be dealing with larger or difficult-to-scale datasets, but also one in which the exact answer isn't required, as an approximate solution would need to suffice to perform the overall task reasonably well.

Due to all of these considerations, we believe that a good place to begin testing the end-to-end applications for this method would be on Image Segmentation tasks via MST clustering. There are already a number of MST methods which perform this task reasonably well [12, 13, 14], with a standard dataset (Berkeley Image Segmentation Dataset [15]). Image processing tasks can be notoriously memory intensive, especially as quality and size increase [17], and as clustering approaches' distance metrics become more elaborate. This would also be an excellent starting point due to the plethora of other well known image segmentation methods that can be used for reference.

We could also now explore changing variables in our initial approach. As mentioned earlier, we have implemented this method with the assumption that we know the full dataset size ahead of time. However, what would happen if we relaxed this assumption and had an unseen dataset? Is it useful to manually set the m and k parameters? These, and other questions that could be answered through varying our assumptions and foundations through developing this method, could and should be the subject of future work.

Finally, we could explore the method's theoretical underpinnings. Specifically, we will attempt to find any provable guarantees about memory consumption and error rate, so as to concretely answer questions such on the method's performance on different types of graphs and its overall reliability. The answers to these questions will guide even further work and point us in the right direction to explore the aforementioned application areas.

7 References

1. Jon Kleinberg and Eva Tardos. 2005. Algorithm Design. Addison-Wesley Longman Publishing Co., Inc., USA.
2. Broder, A. and Mitzenmacher, M., 2004. Network Applications of Bloom Filters: A Survey. Internet Mathematics, 1(4), pp.485-509.
3. Wang, G., Zhang, C. and Zhuang, J., 2014. Clustering with Prim's sequential representation of minimum spanning tree. Applied Mathematics and Computation, 247, pp.521-534.
4. Sabuncu, M., 2006. Entropy-Based Image Registration. Ph.D. Princeton University.
5. Bereg, S., n.d. Applications Of Minimum Spanning Trees.
6. Gera, P., Kim, H., Sao, P., Kim, H. and Bader, D., 2020. Traversing large graphs on GPUs with unified memory. Proceedings of the VLDB Endowment, 13(7), pp.1119-1133.
7. Johnson, Donald B. (December 1975), "Priority queues with update and finding minimum spanning trees", Information Processing Letters, 4 (3): 53–57.
8. The Algorists. 2020. Kruskal's Algorithm : Implementation. [online] Available at: https://efficientcodeblog.wordpress.com/2017/12/11/kruskals-algorithm-implementation. [Accessed 11 December 2020].
9. Eppstein, David (1999). "Spanning trees and spanners". In Sack, J.-R.; Urrutia, J. (eds.). Handbook of Computational Geometry. Elsevier. pp. 425–461.; Mareš, Martin (2004). "Two linear time algorithms for MST on minor closed graph classes" (PDF). Archivum Mathematicum. 40 (3): 315–320.
10. https://pypi.org/project/bitarray/
11. https://pypi.org/project/mmh3/
12. A. Saglam, N.A. Baykan: Sequential image segmentation based on minimum spanning tree representation. Pattern Recognition Letters 87 (2017), pp 155-162
13. Lv, X., Ma, Y., He, X., Huang, H. and Yang, J., 2018. CciMST: A Clustering Algorithm Based on Minimum Spanning Tree and Cluster Centers. Mathematical Problems in Engineering, 2018, pp.1-14.
14. B. Cha, H. Kawano, N. Suetake and T. Aso, "Fast Minimum-Spanning-Tree-like Based Image Segmentation," 2008 Fourth International Conference on Natural Computation, Jinan, 2008, pp. 152-156, doi: 10.1109/ICNC.2008.1.
15. Martin, D., Fowlkes, C. (2001). The Berkeley segmentation database and benchmark. Computer Science Department, Berkeley University. http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/.
16. Starobinski, David; Trachtenberg, Ari; Agarwal, Sachin (2003), "Efficient PDA Synchronization" (PDF), IEEE Transactions on Mobile Computing, 2 (1): 40, CiteSeerX 10.1.1.71.7833, doi:10.1109/TMC.2003.1195150
17. A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen and S. Kvatinsky, "IMAGING: In-Memory AlGorithms for Image processiNG," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 12, pp. 4258-4271, Dec. 2018, doi: 10.1109/TCSI.2018.2846699.