Reproducing *The Power of Pivoting for Exact Clique Counting*

Charun Upara - CSC395

October 21 2020

1 Introduction

Clique counting is one of the most fundamental problems in network science. While a more specific version of the problem - counting 3-cliques or triangles - have been studied and optimized intensively, the problem of counting *all* cliques of *all* sizes is not as deeply researched. To be more precise, there are algorithms for counting all k-cliques in a graph, but most algorithms do not scale well for higher values of k. There are also numerous paths a researcher can take to optimize the k-clique counting algorithms, from devising new sampling algorithm to parallelizing existing algorithms. One such method of speeding up the clique counting process is created by Jain and Seshadhri in the 2020 paper, *The Power of Pivoting for Exact Clique Counting*. [6]

In the paper, the authors propose an exact k-clique counting method that runs significantly faster than previously published algorithms. The authors claim that their algorithm, named Pivoter, is a "scalable, exact algorithm for getting all global and local cliques counts, on real-world graphs with millions of edges."[6]

In this project, I attempted to reproduce the results described by Jain and Seshadhri by implementing their clique counting algorithm, Pivoter, in Julia, and comparing the results and run times to the authors' results described in the paper. To limit the scope of the project, I chose to focus on global clique counting as opposed to also implementing local clique counting.

As will be explored, my implementation has a comparable runtime to the authors', yet it still gives inaccurate countings.

2 Previous Work

As mentioned earlier, clique counting has a long lineage that stems from the simple, recursive solution. The main problem behind this long history is how computationally expensive clique counting is for large graphs. For example, the brute force approach to counting cliques of size k in a graph works by looking at a node and checking whether all of its neighbors, along with all neighbors of neighbors, are connected, but this solution is inefficient, giving an exponential run time. Therefore, various methods of counting and listing cliques have been proposed.

One of the most early and significant result was published by Coenraad Bron and Joep Kerbosch in 1973 [1]. The Bron-Kerbosch algorithm for clique listing works by dividing the vertex set into three sets based on certain properties, which are then used to find cliques. They also proposed a way to optimize the algorithm further in the same paper, which is done by finding a pivot vertex that is guaranteed to not be a part of a clique and therefore making the backtracking more efficient.

The next major result in clique listing was published in 1985 by Norishige Chiba and Takao Nishizeki [2]. The authors proposed an edge-searching strategy that relies on repeatedly removing vertices to quickly arrive at a sparse graph. The run time, expressed in terms of a graph's arboricity $a(G)$, is $O(a(G)m)$. This method is known to be one of the fastest sequential clique listing algorithm that enumerates over all cliques in a graph. The parallel version of this algorithm - kClist - was published on 2018 [3] and extends the quick run time of [2] to larger graphs.

While sequential clique listing algorithms have gotten much faster, the alternate way of arriving at the solution in through the use of sampling algorithms, which in general runs faster but also introduces a certain degree of error. For example, the algorithm devised by Finocchi, Finocchi, and Fusco in [4] is a sampling algorithm. Despite having the same problem as sequential algorithms - impractical run time for large values of k - the algorithm is the fastest sampling algorithm for clique counting. Jain and Seshadhri also published a sampling algorithm prior to Pivoter [5], but again with the caveat that the algorithm could only be used for cliques up to size 10. Other sampling algorithm, with the goal of counting cliques of specific size such as 4 [7] or 5 [8] have also been published, with a basis in sampling wedges [10].

In terms of the theoretical limit of the run time for clique counting algorithms, an important result was published in 2006 that defined the upper-

bound [11], which sets the context for benchmarking different algorithms.

Lastly, the importance of having a fast clique counting algorithm is described in [9]. There are many applications that can be derived from clique counting, such as dense subgraph discovery. A fast algorithm such as Pivoter would allow researchers to discover new properties of large networks that previously could not be considered because of how slow existing clique counting algorithms are.

3 Overview of *Pivoter*

The first major component of the algorithm is a data structured defined as the Succinct Clique Tree (SCT). In a very broad sense, the SCT allows the algorithm to quickly read off clique counts from node and edge labels since it is designed in a way such that every clique in a graph can be expressed as a path in the SCT. This follows from the fact that "every node of the tree (corresponding to a recursive call) corresponds to a subset $S \subseteq V$, and the subtree of calls enumerates all cliques contained in S" [6, p. 3]. Then, the edges in the SCT (called links by the authors) are labeled as vertices "whose neighborhood links to the next recursive call" [6, p. 3]. .

The SCT is defined such that every root to leaf path in the tree corresponds to a clique. However, "not all cliques corresponds to paths" [6, p. 4]. The main insight that the authors developed is that in each root to leaf path, there are links corresponding to pivot calls, which makes the SCT contain a *unique* encoding for all cliques in the graph that are obtained by observing the labels that are defined in the process of building the tree.

After building the SCT, the main work of Pivoter simply becomes extracting clique counts from the SCT by traversing root to leaf paths and incrementing k-clique counts based on the labels of each node and edge of the SCT.

4 Methods

First, I chose to limit the scope of the project to global clique counting, even though the authors also provided implementations for per-vertex and per-edge clique counts. I made this decision in order to focus more on optimizing the runtime of global clique counting, and while it would be an interesting

to run experiments on local counting, there simply isn't enough time for me to implement both.

The main goal for this project was to replicate the correctness and speed of Pivoter by using Julia. To achieve this goal, I looked to the implementation in C that the authors published. C is a natural choice for such a demanding algorithm, since it is easier to control lower-lever details of the algorithm, such as allocating memory, passing variables by reference, optimizing code compilation, and small details such as using 1-indexing instead of 0-indexing. However, a major challenge with the project is that the pseudocode given in the paper is not how the authors actually implemented Pivoter. The pseudocode gives a high-level description that can be used to easiliy prove theorem relating to its correctness and runtime, which also means that many important sub-procedures are not detailed in a way that makes it possible to follow the implementation from the paper alone. The authors also build the SCT recursively in the code as opposed to iteratively as described. Therefore, I based my implementation on the C code given by the authors, with modifications where necessary and where it makes sense to approach the code differently. The exact details of where I modify the code cannot be enumerated, but I will describe the most important parts in order to consider the results of my implementation in context.

To address the problem surrounding 0-indexing in C, I chose to simply move the indexing of every step up by 1. In other words, for all arrays, `arr[i]` in C is equal to `arr[i+1]` in Julia, which also means setting up many loops to iterate from 1 to `arr[length(arr)]`. There were many bugs that result from this switch, since it was challenging to keep track of where to increment a loop one extra time. Because C has the capacity to pass variables by reference to functions, I also needed to adapt my code to always return variables that are modified in functions and update the original variable outside each function. While it is not a major issue, I ran into a problem where my k-clique counter was not being incremented because I didn't properly update the value of two variables in each recursive call. Lastly, the only way to come close to manually deallocating memory in C is to assign a value that does not consume a large amount of memory,such as `nothing` to a variable, which has the tendency to introduce errors in my code as `nothing` might be passed to unsupported functions, so I relied on Julia's automatic memory management in making sure that my program does not consume excessive memory.

The first step in generating an SCT from the input graph is to compute its degeneracy ordering - a step that isn't described in the pseudocode. Because

the authors use their own data structure to store the ordering and base their next steps on this data structure, I chose to port their implementation of `compute_degeneracy_order_array()` into Julia. The procedure works by iteratively removing the vertex with the minimum degree in order to reduce the size of the recursion tree for the SCT. Then for the main procedures, I first attempted to write them from my own understanding of the paper, but because of how much the actual code differs from the implementation, I chose to follow the authors code and make adjustments as needed in order to focus on writing the fastest possible Julia code.

Lastly, I ran the authors' implementation on my computer to compare the runtimes. While their commodity machine is mush more powerful than my personal laptop, I was able to successfully run their code while only taking a few seconds longer than reported on large graphs.

5 Results

My initial implementation was significantly undercounting the number of all clique sizes. But after making adjustments, mostly to array indices and loop conditions, I arrived at the current results, which could be summarized as fast, but still either overcounting or undercounting, depending on the size of the network. I suspect that a combination of small errors in indices and loop conditions compounds into the current inaccuracies, as seen by how minor tweaks to each individual index and condition slightly change the results, but not significantly enough to justify changing the entire structure of the code to match it. While the algorithm is not giving the correct counts, it is still extremely fast.

Even though my results are not accurate, I decided to compare the runtime of my implementation against the authors', since the main motivation behind Pivoter is its extremely fast runtime.

	email-Enron	**web-Stanford**	**web-BerkStan**
n	36692	281903	685230
m	183831	1992636	6649470
Max Clique Size	20	61	201
Authors' Total Count	1.07255302e8	$\sim$2x10^{19}	$\sim$2x10^{60}
My Total Count	1.70610887e9	$\sim$2x10^{22}	$\sim$2x10^{35}

Table 1: Network statistics

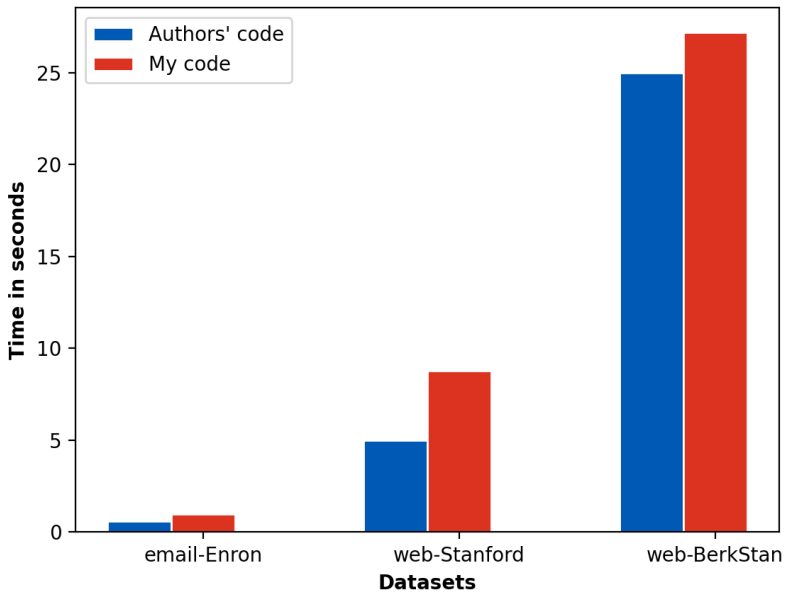

Figure 1: Time taken to find all cliques

Through these three networks, chosen to represent smaller, medium-sized, and larger networks, it can be seen how the runtime of my implementation is comparable to the authors' implementation, even though mine is either undercounting or overcounting the total number of cliques in the networks. As seen in Table 1, my implementation was overcounting in the *email-Enron* and *web-Stanford* networks while severely undercounting in the *web-BerkStan* network, suggesting that the behavior of my code is different in networks with a much higher number of cliques. While the results are obviously inaccurate, I still want to draw attention to the runtime of my implementation. Take the *web-Stanford* network for instance. My implementation was able to count extremely large number of cliques for large values of k while terminating in less than 10 seconds, whereas kClist - the gold standard algorithm - does not terminate on this network [6]. This result suggests how it is possible to generate a much more efficient recursive tree - by building the SCT - for clique listing that can be traversed by the algorithm to efficiently count cliques, even though I was not able to resolve the overcounting/undercounting bug in this particular implementation.

The following outputs from my code and the authors' code, respectively, on the *email-Enron* network shed some light on why the total counts differ.

```
% pivoter.jl data/email-Enron.edges 36692 0
Execution completed in 0.981867229 seconds.
0-cliques: 1.0
1-cliques: 36692.0
2-cliques: 124097.0
3-cliques: 441494.0
4-cliques: 1.663057e6
5-cliques: 5.363435e6
6-cliques: 1.4658e7
7-cliques: 3.4160238e7
8-cliques: 6.8156933e7
9-cliques: 1.16939212e8
10-cliques: 1.73328283e8
11-cliques: 2.22884864e8
12-cliques: 2.4954775e8
13-cliques: 2.439577e8
14-cliques: 2.08651929e8
15-cliques: 1.56285062e8
16-cliques: 1.02500303e8
17-cliques: 5.876305e7
18-cliques: 2.9338323e7
19-cliques: 1.2675008e7
20-cliques: 4.692327e6
21-cliques: 1.467521e6
22-cliques: 380034.0
23-cliques: 79210.0
24-cliques: 12748.0
25-cliques: 1484.0
26-cliques: 111.0
27-cliques: 4.0
Total number of cliques: 1.70610887e9

k, Ck
0, 1.000000
1, 36692.000000
```

```
2, 183831.000000
3, 727044.000000
4, 2341639.000000
5, 5809356.000000
6, 11213163.000000
7, 16985090.000000
8, 20318270.000000
9, 19291746.000000
10, 14604335.000000
11, 8860699.000000
12, 4342925.000000
13, 1742316.000000
14, 582977.000000
15, 165718.000000
16, 40130.000000
17, 8019.000000
18, 1222.000000
19, 123.000000
20, 6.000000

107255302.000000 total cliques
```

The outputs show how my implementation detected 28 sizes of cliques, while the authors' implementation only detected 21. This pattern also exists in the other two datasets. However, even after specifying my algorithm to only find cliques up to size 20, the total count remains close to the previous result, since the count for $k > 21$ is much smaller than the first 20. This result confirms that each individual k-clique count for all k is inaccurate, and therefore resulting in an inaccurate total count.

6 Conclusion and Future Work

The goal of this project was to reproduce the results of [6], and while I was not able to arrive at the same results, this project still shows how it is possible to implement Pivoter in Julia and get the promised, fast runtime, even on graphs that highly known algorithms do not terminate. The inaccuracies in my results most likely stem from my own errors in translating C code to

Julia code, which suggests that future work in this project could result in a complete reproduction of the authors' results in Julia.

While fixing the counting errors has the highest priority for future work, there are numerous other paths to take to extend this project further. One of those paths is to implement local clique counting, which has applications in studying structures of large networks that were simply not possible without a fast algorithm. There, most likely, are also ways to reduce the memory usage of my Julia code, which would result in a more efficient algorithm overall.

References

[1] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[2] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[3] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*, pages 589–598, 2018.

[4] Irene Finocchi, Marco Finocchi, and Emanuele G Fusco. Clique counting in mapreduce: Algorithms and experiments. *Journal of Experimental Algorithmics (JEA)*, 20:1–20, 2015.

[5] Shweta Jain and C Seshadhri. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th International Conference on World Wide Web*, pages 441–449, 2017.

[6] Shweta Jain and C Seshadhri. The power of pivoting for exact clique counting. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, pages 268–276, 2020.

[7] Madhav Jha, C Seshadhri, and Ali Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th International Conference on World Wide Web*, pages 495–505, 2015.

[8] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1431–1440, 2017.

[9] Ahmet Erdem Sariyuce, C Seshadhri, Ali Pinar, and Umit V Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 927–937, 2015.

[10] C Seshadhri, Ali Pinar, and Tamara G Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(4):294–307, 2014.

[11] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical computer science*, 363(1):28–42, 2006.