GRAPHBLAS: SOLVING GRAPH ALGORITHMS WITH LINEAR ALGEBRA

An Undergraduate Research Scholars Thesis

by

JULIO CESAR MALDONADO GUZMAN

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Timothy Davis

May 2019

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

GraphBLAS: Solving Graph Algorithms with Linear Algebra

Julio Cesar Maldonado Guzman
Department of Computer Science and Engineering
Texas A&M University

Dr. Timothy Davis
Department of Computer Science and Engineering
Texas A&M University

GraphBLAS is a C library written by Dr. Davis that allows users to easily represent graphs as sparse matrices. GraphBLAS also allows linear algebra operations on its graphs, so that users can develop graph algorithms in the language of linear algebra. Concluding that GraphBLAS is a more efficient and concise way of writing graph algorithms is important to academia, as it'd introduce a better approach for researchers and students to learn and write graph algorithms. The ability to write graph algorithms efficiently will allow researchers to test what they're needing to do at a quicker pace. Instructors will also be able to teach and explain graph algorithms to their students in a way that they can easily grasp the material. In return, the students will get to learn the material in a new way and be able to test their understanding. My outcomes will further the validation and understanding of GraphBLAS as an alternative to regular graph algorithms. Furthermore, such graph algorithms will also allow for software developers in industry to write graph algorithms quickly. Such algorithms are crucial to various situations such as figuring out bots on Facebook and search results on Google. Kepner and Gilbert prefaced that graph

algorithms “have become essential in controlling the power grid, telephone systems, and, of course, computer networks (xxv)”, further validating how impactful a new way to compute these algorithms could be. Previous research on this topic conducted by Buluc and Gilbert detail how to approach many different graph algorithms in the language of linear algebra. That research can be referred to gather information on how to better approach graph algorithms in GraphBLAS. We will be able to compare how easy and efficient it is to write such algorithms to the regular method using adjacency matrix or vertexes to test my research statement of GraphBLAS being a powerful and expressive way to develop graph algorithms.

DEDICATION

Para Bri, que ha estado ahí para mí para todo. Para mi hermano, de quien no podría estar más orgulloso. Para mi madre, que siempre nos dio todo lo que tenía para criarnos. Para mi padrastro, quien nos enseñó a trabajar duro y no esperar nada a cambio.

For Bri, who has been there for me for everything. For my brother, who I couldn't be prouder of. For my mother, mi mamá, who raised my brother and I no matter how hard it got. For my step-father, mi papá, who taught us how to work hard and not expect anything in return.

ACKNOWLEDGEMENTS

I'd like to thank Dr. Timothy Davis for being my faculty advisor, mentor, and role model. I've learned a lot in the past year and I've seen firsthand how an idea can become reality through determination and hard work. Thank you for the guidance and support throughout the course of this research.

Thanks also to my peers, colleagues, and department faculty and staff for making my time at Texas A&M University a great experience. I'd like to personally thank Dr. Teresa Leyk for her support throughout my time at A&M. Thank you for letting me become a peer teach to pass off my knowledge to other students who struggled just like I did. Thank you also to Dr. Scott Schaefer and Dr. Dilma Da Silva for doing everything you could to help me out.

Finally, thanks to all my friends who have supported me, no matter how small or big of a part each of you have played. I am forever grateful.

CHAPTER I

INTRODUCTION

Linear algebra can be used for several applications, ranging from mechanical vibrations to computer vision. It's versatility and powerful operations stirred the idea of using linear algebra to solve graph algorithms. The GraphBLAS community came around and rallied behind this concept and began to contribute to graphblas.org. Dr. Timothy Davis went on to develop the largest, active and open source library in C, implementing graph algorithms solved with linear algebra appropriately called GraphBLAS.

Why Use Linear Algebra

Math and computer science often go hand in hand in educational settings. A solid mathematical background is often considered crucial to doing well in the field of computer science. However, computer science usually begins to deviate from math and becomes about quickly developing efficient code. Efficient code is where we introduce math once again, i.e. using bit shifting for quicker multiplications, Fermat's theorem to determine whether a number is prime, or using linear algebra to perform a BFS on a graph. Linear algebra has powerful operations and expressions that allow for easy to understand and efficient operations on a graph.

Why Not to Use Linear Algebra

Some might fear the idea of using linear algebra rather than traditional data structure, such as queues and adjacency matrixes, to perform graph algorithms. It can sound daunting at first, especially to those who aren't comfortable with math. The barrier to entry might be too high. Another potential reason is that there are fewer resources to turn to. If a company decides to switch over, then they would have to refactor a substantial amount of code; and if the

company struggles with a complex algorithm, they have less options to receive help. These issues might be overwhelming at first, so figuring out whether switching over is worth it is crucial.

Moving Forward

There are advantages and disadvantages for using GraphBLAS – I'll list out as many as I can find so that I provide insight into how powerful the library is. It is important to keep in mind that although the GraphBLAS library is intended for mass usage for everyday problems, it will not always be the best solution. Some companies have already developed their own custom graph libraries for their business, while researchers might enjoy using a particular graph library, and students may not care too much about efficiency. Choosing a graph library is a conscious decision that considers many factors. That's standard in computer science. GraphBLAS is not intended to be the solution to every problem, rather it is intended to raise the floor so that every developer can immediately start representing and manipulating graphs in a very efficient manner.

CHAPTER II

METHODS

I am the first documented undergraduate to experiment with the SuiteSparce GraphBLAS library. The three factors I will talk about will be my ease of understanding, ease of writing, and efficiency of the algorithms. These three factors will be a good indicator as to how simple it is to learn the library, how easy it is to use the library, and how good the library is. Nevertheless, my experiences are personal and somebody else might have an easier, or more difficult, time with GraphBLAS.

Ease of Understanding

GraphBLAS is built on the idea of using linear algebra to solve graph algorithms, so many who struggle with linear algebra might struggle with understanding what the function calls do. I have a mathematics minor where I've taken a variety of math courses, including linear algebra. I'll share what I think is important to know in linear algebra so that anybody who does not have a similar background can be prepared to start using GraphBLAS. These metrics are important in understanding what it'll take to transition to GraphBLAS.

Ease of Writing

To understand is one thing, but to use is another. The built-in functions of GraphBLAS should be simple enough to understand and use so that a developer can start using them after reading the API. The flow of the program should also be easily understood, so that other developers can read the code and know what's happening. These two factors are important in software development for companies, research for academics, and learning for students.

Efficiency

I plan on implementing three graph algorithms: BFS, Floyd-Warshall, and Prim's algorithms in C, Python, and in GraphBLAS. I'll then be able to compare the run times of the algorithms for various graphs, from dense to sparse, to determine whether scenarios exist where a different approach is preferred. Through these measurements, I hope to increase the understanding of what it will take to take GraphBLAS to a wider audience, including education, academia, and industry.

A Reliable Experience

Although these factors aren't perfect, they are commonly used by software development teams, research teams, and students to decide on what approach to use to solve a problem. The simpler it is to learn something, then use it, and get a very good outcome from it often corresponds to a better experience. A good experience is what will later create active users, contributors, and advocates for GraphBLAS. We can then reliably use these metrics to determine the feasibility of GraphBLAS.

CHAPTER III

RESULTS

There are many factors that influence how easy it is to grasp a new technology. Similar experiences are probably one of the most important. If I've worked with a functional programming language before, then I'll probably have an easier time learning a new one. GraphBLAS is completely written in C, and most of my programming experience involves using C++. I've learned various graph algorithms, however learning them under the context of linear algebra was a new challenge.

Ease of Understanding

To start off, GraphBLAS is not the easiest library to learn. Some of the challenges of learning the library include lack of references that are easy to find, knowledge of C, and knowledge of the library itself. However, most of the issues can be resolved through more support and usage of the library.

Knowledge

There exists a large online community of GraphBLAS supporters, but what's important to note is that there is not an official GraphBLAS implementation or single library that is widely supported. Dr. Davis' version of GraphBLAS is one of the largest and most complete libraries available, while Kepner and Gilbert's "Graph Algorithms in the Language of Linear Algebra" is one of the most encompassing books written about how to solve such algorithms. Most of the algorithms implemented are based off the pseudocode and explanations written in the book. It's therefore safe to say that there are many experts in GraphBLAS, but not many experts in Dr. Davis' implementation of GraphBLAS. Furthermore, one of GraphBLAS strong points is the

ease of parallelization. Parallelization is an advanced topic, and being able to efficiently implement it is hard. Knowledge of parallelization, the C library, and GraphBLAS are all necessary to get the most out of the algorithms.

Resources

Many of the libraries that I have used are very popular and widely used. Through mass and wide spread usage arise experts and beginners. Beginners will be able to ask introductory level questions, while experts will be able to answer them. Experts can then go on to create blogs, APIs, and tutorials on how to best use a library. I was lucky enough to have Dr. Davis, the creator of GraphBLAS, as my advisor, so I had the best resource available. However, when asking a more trivial question, it might be easier to simply search for it online and have examples to go off. This short-term problem can of course be fixed as experts arise.

Ease of Writing

Writing efficient and easy to understand algorithms is always a difficult task. There are always opportunities to convolute an algorithm making it slower or harder to read. Let's consider Floyd-Warshall as an example.

Floyd-Warshall

Here is a snippet of the main component of the algorithm:

```
for (int k = 0; k < m; k++) {
   for (int i = 0; i < m; i++) {
      for (int j = 0; j < m; j++) {
         if (g[i][k] == INT_MAX || g[k][j] == INT_MAX)
                  continue;
         if (graph[i][j] > graph[i][k] + graph[k][j])
            graph[i][j] = graph[i][k] + graph[k][j];
      }
   }
}
```

The algorithm is intended to find the shortest path for every single pair in a graph. Therefore, you need to run through the entire graph and check if there is a path from one entry to another

entry with a cost less than the current path. If there is, then you can update the cost to the cost of the better path. It is also wise to set entries that don't have a path to another entry with the value of INT_MAX, which is used to represent the largest value an integer can be. This INT_MAX also denotes infinity, meaning that the cost to get to that entry in the graph is infinite, thus accessing it is impossible. We must check if either of the added entries are INT_MAX to ensure that overflow does not occur so that we calculate the correct cost. This is an advanced algorithm that runs in $O(m^3)$. Since the graph must be m x m size, we must check every single item in the rows and columns, which is of size m x m thus we must perform m x m operations. We then have to run through every entry in the rows to update the path cost if necessary, so we perform another m operations, ultimately equating to m x m x m operations. The triple for loop is hard to understand and debug if not implemented correctly. An algorithm like this can, however, be easily written in GraphBLAS.

Floyd-Warshall in GraphBLAS

Here is a snippet of the code to implement Floyd-Warshall in GraphBLAS:

```
for (GrB_Index i = 0; i < m; i++) {
        GrB_extract(A, GrB_NULL, GrB_NULL, G, GrB_ALL, m, &i, 1, GrB_NULL);

        GrB_extract(B, GrB_NULL, GrB_NULL, G, &i, 1, GrB_ALL, m, GrB_NULL);

        GrB_mxm(C, GrB_NULL, GrB_NULL, GxB_MIN_PLUS_FP32, A, B, GrB_NULL);

        GrB_eWiseAdd(G, GrB_NULL, GrB_NULL, GrB_MIN_FP32, C, G, GrB_NULL);
}
```

This algorithm does the exact same thing as the previously described algorithm, except it uses linear algebra to compute the shortest paths. The code still needs to iterate through all of the entries in the graph, hence the outer for loop. It then extracts the next row to A and column to B to add the pairs and place them in the temporary graph C. The code will next update the values of the graph G with the smaller value, whether it be in the graph G or the computed value in C. The

OK function seen here is a C macro that will ensure the GraphBLAS functions were called and executed correctly. If the function fails, then the OK macro will be able to cleanly display what went wrong internally.

Comparison

It's easy to see that the GraphBLAS version is easier to read, write, and understand. Floyd-Warshall is very easy to represent in the language of linear algebra because of the nature of it. Other algorithms such as Prim's or DFS are very hard. The built-in functions that GraphBLAS provides almost even force the developer to write good clean code. Since we know what the functions do, it is easy to read them and understand them. For example, we know that the GrB_eWiseAdd function will perform an operation on every entry in 2 separate graphs and then place the result into a separate graph. We can decide on what operation, such as plus, minus, division, etc., so it's easy to understand what's going on when we see it. On the other hand, reading the for loops in the regular version and then the change of variables in the line "graph[i][j] > graph[i][k] + graph[k][j]" is confusing. It's not obvious that you're comparing the current cost to the cost of a different path. This is a useful example to see when GraphBLAS is the better option.

Efficiency

This is likely the most important factor. Many systems, such as Facebook, Google, and more, absolutely need their algorithms to be as fast as possible. Researchers might not absolutely need speed, but if they can quickly read in a graph, compute important operations on it, and then quickly get those results, they are likely to take notice. Students are the least likely to prioritize efficiency, since speed is not usually necessary. Nevertheless, efficiency encompasses how much space and real time the algorithms use up. We'll consider Floyd-Warshall and then BFS.

Floyd-Warshall Efficiency Comparisons

In order to compare the two algorithms fairly, I used a variety of different graphs to test against each other. I then ran the algorithms 9 times each and grabbed the medium, since the average could be skewed by outliers. Before I compared the run times, I had to ensure that both versions of the Floyd-Warshall were correct. In order to do that, I grabbed 3 different graphs from https://sparse.tamu.edu/, an online matrix collection developed by Dr. Davis, and ran the regular and GraphBLAS version of Floyd-Warshall on them. I then wrote the resulting graph to separate files and compared them. If they were equivalent, then the algorithms were correct. The graphs used were of size 1000 x 1000, 2003 x 2003, and 2500 x 2500 and each were of various density. The results were the same, so I concluded that both algorithms were correctly implemented.

Floyd-Warshall Efficiency Results

I first tested the 1000 x 1000 graph of various densities. It's clear to see, even on a smaller graph such as this, on Table 1 the speed up of GraphBLAS as the graph becomes sparser. It's also clear to see that it slows down as the graph becomes denser. At its densest, when it's full, the regular version is 5.5 times faster. At the sparsest attempted, with only 749 entries, the GraphBLAS version is 80.1 times faster. The graph was only 0.075% full.

Table 1 - Results of Floyd-Warshall on a 1000 x 1000 graph

Entries	GraphBLAS	Regular
749	0.023959 s	1.920418 s
1,498	1.429844 s	2.106021 s
3,996	12.191048 s	3.441595 s
1,000,000	30.538794 s	5.487051 s

Let's now consider the 2003 x 2003 graph in Table 2. We see the same pattern on this graph too. At its sparsest, the GraphBLAS algorithm is 140 times faster. When the graph is full with over 4 million entries, the regular version is 11 times faster.

Table 2 - Results of Floyd-Warshall on a 2003 x 2003 graph

Entries	GraphBLAS	Regular
815	0.113640 s	15.954680 s
1,629	0.444868 s	16.005836 s
3,258	16.137870 s	19.204371 s
23,973	31.683506 s	20.750747s
4,012,009	255.625070 s	22.238127 s

For the 2500 x 2500 graph, the same pattern is found for both algorithms as seen in Table 3. The difference here is that the increase in runtime is very noticeable. The regular version is quite consistent in speed, until it processes a half full graph. It takes 10 more seconds. To process the full graph takes over twice as long. To process it in GraphBLAS is 7.2 slower. On the other hand, at the sparsest attempt, the GraphBLAS version is 1200 times faster.

Table 3 - Results of Floyd-Warshall on a 2500 x 2500 graph

Entries	GraphBLAS	Regular
183	0.026131 s	31.347659 s
2,914	3.534615 s	30.024570 s
927,322	44.652251 s	34.654181 s
1,854,643	77.027950 s	39.839372 s
3,123,750	130.489275 s	43.648497 s
6,250,000	646.821772 s	90.459550 s

Floyd-Warshall Takeaways

The data presented is impressive. GraphBLAS slows down as the graph becomes denser, but never by an extremely high amount. It does however become far faster as the graph becomes sparser, all the way up to 1200 times as fast. The above data doesn't show the memory usage, but it is crucial to note that the regular version will always allocate m x m memory, while GraphBLAS will only allocate just enough memory to hold the entries. Some programs that have high memory requirements will be unable to be completed using the regular version due to constraints by the machine it's running on. GraphBLAS would be able to compute the algorithms on far more graphs thanks to its clever memory usage. GraphBLAS is a better choice here.

Breadth First Search

Let's now consider the breadth first search (BFS) algorithm, which traverses a graph until some condition is met. Here is a snippet of the main component of the code:

```
while (!isEmpty(&q)) {
        pop(&q, &sourceNode);

        for (int i = 0; i < m; i++) {
                if (!visited[i] && graph[sourceNode][i] != INT_MAX) {
                        visited[i] = true;
                        push(&q, &i);
                }
        }
}
```

The BFS algorithm will usually require a queue to keep track of what entries need to be visited and a visited array to keep track of what entries have been visited. While the queue is not empty, we'll pop from it and store the value in sourceNode. We'll then go through its neighbors to check if they have not been visited. We are also checking that it's possible to visit the neighbor, that is the cost to access it is not infinite. If both conditions have been met, then we'll mark it off as visited in the array and push the neighbor on to the queue. This will let us traverse

the tree starting at whichever entry we decide until no more neighbors are accessible. This algorithm will run in O(m) time since we at most access every entry in the tree once, if every entry is accessible from the source entry.

BFS in GraphBLAS

Here is a snippet of the code:

```
for (int64_t level = 1; successor && level <= max_level; level++) {
    GrB_assign(v, q, NULL, level, GrB_ALL, n, NULL);

    GrB_vxm(q, v, NULL, LAGraph_LOR_LAND_BOOL, q, A, desc);

    GrB_reduce(&successor, NULL, LAGraph_LOR_MONOID, q, NULL);
}
```

Here, q represents the entries visited at each level and v the result vector of each iteration. The first step is to assign v to the level using q as the mask and then updating q to all the unvisited entries using !v as the mask. We then update the successor – if there are no new nodes to visit, then it is false, thus the for loop will terminate. The max_level will usually be m, where the graph is of size m x m, so that the entire graph can be checked. This algorithm should run in O(m), however that will depend on the implementation of the three functions called. If the graph is very sparse, they would be done very quickly. Otherwise, they can be quite slow.

Comparison

The regular version here is easier to read, write, and understand. BFS is a bit harder to represent in the language of linear algebra because of the operations required. A queue is often the first data structure that a student will learn so it's easy to see why many would like to use a queue here rather than linear algebra. However, C does not have built in data structures, so I had to write my own queue data structure, which took more time than writing the code necessary to perform the BFS. The functions necessary on GraphBLAS to perform BFS are built-in and, once

again, almost force the user to write clean code. Still, this is a practical example to see when GraphBLAS is not the superior option.

BFS Efficiency Comparisons

I used the same testing methods to determine that the BFS implementations were correct as for Floyd-Warshall. This required me to run both algorithms on the same graphs and compare the outputs to each other. If they were the same, then I had properly tested that the algorithms worked. In order to grab the practical runtime, I also ran each algorithm 9 times and grabbed the medium to ignore the outliers, just as before.

BFS Efficiency Results

The BFS algorithm is quite fast, unlike the Floyd-Warshall algorithm. This explains the fast runtimes for a 1000 x 1000 graph in Table 4 – not a single run being slower than a second. Nevertheless, we can see that the regular BFS is notably faster than the GraphBLAS BFS, even on a very sparse graph. The regular version is 3.8 times faster with only 749 entries, but 17 times faster when it's completely full.

Table 4 - Results of BFS on a 1000 x 1000 graph

Entries	GraphBLAS	Regular
749	0.000166 s	0.000043 s
2,245	0.008284 s	0.002478 s
6,733	0.016305 s	0.002518 s
1,000,000	0.041791 s	0.002452 s

The same pattern exists for the 2003 x 2003 graph in Table 5. At its sparsest, the regular BFS is 76 times faster, but when it's full it's 108 times faster.

Table 5 - Results of BFS on a 2003 x 2003 graph

Entries	GraphBLAS	Regular
815	0.009140 s	0.000119 s
2,445	0.027511 s	0.009075 s
7,335	0.026504 s	0.010535 s
4,012,009	1.086717 s	0.010062 s

The results here are slightly unexpected for the 2500 x 2500 graph. With only 183 entries, the regular BFS is 1.1 times faster. When the graph is full, the regular BFS is 16 times faster. This might be because of useful caching or a built-in speedup from GraphBLAS.

Table 6 - Results of BFS on a 2500 x 2500 graph

Entries	GraphBLAS	Regular
183	0.001498 s	0.001304 s
549	0.004704 s	0.003517 s
1,647	0.016576 s	0.011196 s
6,250,000	0.241566 s	0.015115 s

BFS Takeaways

This algorithm does highlight a few of the downfalls of GraphBLAS. The algorithm does appear to speed up as the graph becomes sparser, but not notably. Even when the graph is extremely sparse, the GraphBLAS BFS could not compete with the regular BFS. This is the case because the vector v – it's sparse at first, but slowly becomes full. The memory required by the algorithm is also roughly equivalent, so there's not a notable improvement there either. However, I did use the matrix reader that is readily available in GraphBLAS in my regular BFS

implementation and I had to create my own Queue data structure in C. Reading in a file, parsing that file line by line, and creating a queue data structure are, at the very least, tedious, if not difficult. It is important to note that GraphBLAS could be improved here. The next version of GraphBLAS will have a parallelized BFS readily available. Writing a BFS in parallel is very difficult and time consuming to implement, so a parallelized BFS ready to be used is attractive. It will also be far faster than a non-parallelized version. Nevertheless, there will be algorithms that GraphBLAS will be the superior option and others where a regular approach is preferred. There's always a trade-off.

CHAPTER IV

CONCLUSION

GraphBLAS is a powerful and expressive way to write graph algorithms. The learning curve can be steep, but the process is rewarding. The overall experience of learning, understanding, and using the library is worth it. It's clear that the library was developed with a lot of thought and effort put into it to ensure extreme versatility. With such versatility, the algorithms can be extended to consider various factors such as getting the parents from the Floyd-Warshall algorithm, creating the path to obtain an entry in a BFS, and more. GraphBLAS even feels different than other graph libraries because of how apparent it is that it considers practical applications. It is cautious to only use the required space and speeds up drastically for sparse graphs, which most graphs are. These factors contribute to the overall confidence in GraphBLAS being a reliable and efficient way of representing graphs and computing graph algorithms

Future

Although GraphBLAS is a great library, it can be hard to quickly write your own graph algorithms. This is why LAGraph is in development. LAGraph is built on top of GraphBLAS to provide easy to use functions that will run optimized and parallelized algorithms such as Floyd-Warshall, BFS, and K-truss in a single line. These functions will provide all the power, efficiency, and cleverness of GraphBLAS with an improved API that allows for simple calls with big results. These two libraries are viable and can gain traction to develop a new standard in solving graph algorithms.

REFERENCES

Buluc, Aydin, et al. *The GRAPHBLAS C API Specification*. 2018

Kepner, Jeremy V., and John R. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematic, 2011.

Timothy Davis. *User Guide for SuiteSparse: GraphBLAS*. 2018.