MULTI-SOURCE BREADTH FIRST SEARCH IN MATRIX NOTATION

An Undergraduate Research Scholars Thesis

by

ALEXANDRA GOFF

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

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Timothy Davis

May 2023

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Alexandra Goff, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Multi-Source Breadth First Search in Matrix Notation

Alexandra Goff
Department of Computer Science
Texas A&M University

Faculty Research Advisor: Dr. Timothy Davis
Department of Computer Science
Texas A&M University

In this thesis, I will discuss a multi-source breadth first search algorithm I wrote for LAGraph. It allows a user to get the BFS parent and level data of a graph for several source nodes at once instead of having to do each source individually. This is not only easier on the user, but because of the parallelization that the matrix representation allows it is also more efficient than looping through each of the nodes of interest. While this is valuable to a user in its own right, a multi-source breadth first search also opens the door to other algorithms. I highlight methods of estimating or directly obtaining diameter and discuss further algorithms that could be added in the future.

DEDICATION

To my family, friends, instructors, and peers who supported me throughout the research process.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Timothy Davis for his guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

The graphs analyzed for Multi-Source Breadth First Search algorithm evaluation were provided by Professor Davis.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Undergraduate research was supported by Intel, NVIDIA, Redis, MathWorks, and Julia Computing at Texas A&M University.

1. INTRODUCTION

1.1 Background

1.1.1 Breadth First Search

A breadth first search is a method of searching through a graph starting from a source node and expanding out to each node adjacent to it in levels. The primary information usually gathered about a node in a breadth first search are its level and its parent. A node's level indicates how far it is from the source node, where the source node is level 0, nodes adjacent to the source node are level 1, nodes adjacent to level 1 that are not already in a level become level 2 and so on. A node's parent is a node on the level above the node of interest from which the node was found. If a node has two or more nodes that could be its parent, any of the potential parents can be considered a valid result. The source node of a breadth first search is considered to be its own parent. A breadth first search is completed when the highest level has no adjacent nodes that are still outside the breadth first search and results of a breadth first search are often displayed in a tree with the source node at its root.

1.1.2 Diameter definition

A graph's diameter is the graph's maximum eccentricity, where the eccentricity of a node is the maximum distance from that node to all other nodes in a graph [1]. Nodes that exhibit this eccentricity are called peripheral nodes. Another way the definition can be phrased is that a graph's diameter is the longest shortest path in the graph [2]. Defining diameter this way is fairly straightforward for graphs in which all the nodes are connected. However, some, such as [2], caveat that a disconnected graph has an infinite diameter. For the purposes of this paper, I will be

treating the diameter of a graph as the longest shortest traversable or connected path and not detecting disconnected graphs.

1.1.3 GraphBLAS and LAGraph

GraphBLAS is a package that seeks to standardize the representation of graphs in the language of linear algebra and facilitate the use of this representation [3]. It includes methods of representing a graph in matrix form and matrix operations that can be used to create algorithms around those graphs. It also facilitates the creation of matrices and vectors to interact with these graphs. LAGraph is a library of algorithms that use GraphBLAS [4]. LAGraph contains many algorithms for working with and analyzing graphs efficiently based on the advantages of working with the linear algebra representation of the graph.

1.1.4 Semirings and Masks

One key concept I had to become familiar with for this project was semirings. Semirings are a way of defining operations performed on matrices and are composed of two monoids. The semiring most commonly used is the one used for matrix multiplication, which would be a times-add semiring. This refers to how the initial values are multiplied together, then those products are summed, with multiplication and addition as the two monoids. To enable better matrix operations, GraphBLAS allows for other types of monoids, such as ones that just determine if a value is present using or or and boolean logic or ones that identify the row or column a value is from.

Another tool used to improve matrix computations in GraphBLAS are masks. A mask is used to decide which elements in an output matrix to actually compute. For example, in matrix multiplication, if a mask is used and disallows filling in of C(i, j) in output matrix C, then in the

output matrix, C(i, j) will not be filled in. In my project, this is primarily used to ensure that previous levels of the breadth first search trees are not overwritten by later levels.

1.1.5 Project Goal

In my project, I aimed to build and test an algorithm to extend the functionality of LAGraph's current Breadth First Search algorithms by developing a multi source breadth first search algorithm that takes advantage of the matrix notation of LAGraph to more efficiently perform a breadth first search from each node of interest at once, rather than just running a breadth first search for each node individually. I then developed two algorithms based on the multi-source breadth first search for determining the diameter of a graph. One to find the exact diameter and the other to estimate it.

2. METHODS

2.1 Multi Source Breadth First Search Algorithm

2.1.1 Base Algorithm and Similarities

In building my algorithm I based it off of an existing breadth first search algorithm in LAGraph [5]. That initial algorithm took in a single source node and a graph containing that source node and generated vectors that gave the parent node and level of each node. The algorithm had options to record just parent or just level vectors or compute both parent and level vectors.

The option to compute parent, level, or both parent and level information was carried over to my algorithm. Also carried over was the main structure of how the algorithm generates the parent and level information. The algorithm starts with variable initialization and verification of inputs. In particular, it verifies that it has been asked to compute at least one of parent and level, since while the algorithm could do nothing and return success if not asked for either level or parent information, we determined that if the algorithm was called it was likely expected to do something, so being called to do nothing was likely an error that should be raised to the user. There is also verification that the source or sources for the breadth first search are in the graph. After the verification, there is further initialization set up based on whether the user has requested parent information, level information, or both. Then the main loop for the breadth first search is entered and parent, level, or parent and level information is generated and stored. Once the loop is complete, parent and level information are made available to the user, all created variables are cleaned up, and success of the algorithm is returned.

2.1.2 Algorithm Adaptations

The main adaptation to the algorithm in my project was adding a dimension to the variables in the algorithm. This means that instead of a single source node I have a vector of source nodes and instead of vectors for the level and parent of each node I have matrices of the level and parent of each node with respect to each source node. The advantage of this dimension increase is that I can compute each iteration of level and parent information for all the source nodes at once instead of only for one source node at a time.

One other adaptation from the original algorithm was a simplification of the algorithm's method of advancing the frontier. In the original algorithm, for efficiency, the algorithm would advance the frontier in either a push method or a pull method. Push method advancement is done by starting from the nodes in the frontier and looking at all nodes adjacent to them to find any that have not yet been seen and add them to the next level, pushing the frontier out. Pull method advancement, by contrast, looks at all unfound nodes and determines whether any of them has an adjacent node in the frontier and if so adds the node to the next level. In the original algorithm, switching to the pull method towards the end of the program, when the frontier is large and the remaining unfound node pool is small, was used to increase the algorithm's efficiency. However, since the push advancement method is more efficient for most of the process and it would be harder to efficiently decide between methods with multiple breadth first searches occurring in parallel, I have developed my algorithm to always use push method advancement.

2.2 Diameter: Exact and Estimated

2.2.1 Exact Diameter Calculation

If one performs a breadth first search on every node in a graph, the graph's diameter will be the highest level found of all the breadth first search trees. For my exact diameter algorithm, I

break the nodes into groups, grouping by a variable k that the function takes in and loop through all the nodes calling a multi-source breadth first search on k nodes at a time. Breaking the nodes up is done to keep the space requirements of the algorithm manageable at the cost of losing some of the parallelization. For each batch, I store each node's eccentricity, that is, the highest level in its breadth first search tree. Once all the breadth first search levels have been found, the highest level is the graph's diameter. A vector of peripheral nodes, the nodes that are at that maximum level from another node, can also be returned.

2.2.2 Diameter Estimation

For the estimation of the diameter, I again use a passed in k value to keep to a manageable size. In that algorithm, I start with k random initial nodes and run a breadth first search on them. Then I take the node(s) with the highest level in the found breadth first search trees and run a multisource breadth first search with k of those nodes, or all of them if there are fewer than k nodes at the maximum level. This is repeated until the maximum level of the current breadth first search tree is the same as the maximum level of the last iteration. While this algorithm is not guaranteed to find the diameter of a graph, it does find a reasonable lower bound for it. Like the exact diameter algorithm, this algorithm can also return the nodes found with this highest breadth first search level, though in this case the nodes are pseudo peripheral instead of just peripheral nodes because they are only at a guessed boundary, not a certain one. The algorithm does have an initial check for if k is greater than or equal to the total number of nodes in the graph so that it doesn't have to do a second search of the graph, since that high a k value makes it the exact diameter algorithm.

2.3 Pseudo-code Multi Source Breadth First Search Algorithm

Algorithm 1: Multi-Source Breadth First Search

Input: Matrix* level, Matrix* parent, Graph G, Vector src
Output: Returns GrB_SUCCESS or an error code

```
1   // variable creation and input checking
2   Matrix frontier = NULL;       // current frontier
3   Matrix pi = NULL;     // parent matrix
4   Matrix v = NULL;      // level matrix
5
6   bool compute_level  = (level != NULL) ;
7   bool compute_parent = (parent != NULL) ;
8   if (compute_level ) (*level ) = NULL ;
9   if (compute_parent) (*parent) = NULL ;
10
11  // get problem size
12  Matrix A = G->A;
13  Index nsrc; // holds the number of sources
14  Index n;
15  GrB_Matrix_nrows (&n, A) ;
16  GrB_Vector_size (&nsrc, src) ;
17
18  Semiring semiring;
19
20  if (compute_parent)
21  {
22          // semiring used for parent computation
23          semiring = GxB_ANY_SECONDI_INT*
24          // set up parent matrix
25          pi = Matrix[nsrc][n];
26          // parent for each source is itself in its corresponding row
27          for (int64_t s = 0; s < nsrc; s++)
28          {
29                  Index currsrc = src[s];
30                  pi[s][currsrc] = currsrc;
31          }
32          // set up frontier
33          frontier = Matrix[nsrc][n];
34          // frontier also starts with the source nodes in their rows
35          for (int64_t s = 0; s < nsrc; s++)
36          {
37                  Index currsrc = src[s];
38                  frontier [s][currsrc] = currsrc;
```

```
39          }
40  }
41  else
42  {
43          // semiring for when only level is needed
44          semiring = LAGraph_any_one_bool;
45          // when only getting the level, the frontier can just be booleans
46          frontier = Matrix[nsrc][n];
47          // frontier starts with the source nodes in their rows being true
48          for (int64_t s = 0; s < nsrc; s++)
49          {
50                  Index currsrc = src[s];
51                  frontier[s][currsrc] = true;
52          }
53  }
54  if (compute_level)
55  {
56          // set up level matrix, sources are denoted by level 0
57          v = Matrix[nsrc][n];
58          for (int64_t s = 0; s < nsrc; s++)
59          {
60                  Index currsrc = src[s];
61                  v[s][currsrc] = 0;
62          }
63  }
64
65  // core BFS traversal and labeling
66  Index nfrontier = nsrc;          // number of nodes in the current level
67  Matrix mask = (compute_parent) ? pi : v ;
68  for (int64_t nvisited = nsrc, k = 1 ; nvisited < n*nsrc ; nvisited += nfrontier, k++)
69  {
70          // advance the frontier
71          GrB_mxm (frontier, mask, NULL, semiring, frontier, A, GrB_DESC_RSC);
72          // stop if the frontier is empty
73          GrB_Matrix_nvals (&nfrontier, frontier) ;
74          if (nfrontier == 0)
75          {
76                  break ;
77          }
78          // assign parents/levels
```

```
79          if (compute_parent)
80          {
81                  // frontier(s, i) currently contains the parent id of node i in tree s.
82                  // pi{frontier} = frontier, where {frontier} means using frontier as a
    mask
83                  GrB_assign (pi, frontier, NULL, frontier, GrB_ALL, nsrc,
84                  GrB_ALL, n, GrB_DESC_S) ;
85          }
86          if (compute_level)
87          {
88                  // v{frontier} = k, the kth level of the BFS
89                  GrB_assign (v, frontier, NULL, k, GrB_ALL, nsrc,
90                  GrB_ALL, n, GrB_DESC_S) ;
91   }
92  }
93  // clean up and return results
94  if (compute_parent) (*parent) = pi ;
95  if (compute_level ) (*level ) = v ;
96  return (GrB_SUCCESS) ;
```

While the function only returns a value indicating its success or an error message, it also uses the passed in “parent” and “level” matrix pointers to basically return the created parent matrix, level matrix, or both matrices to the user as well. Parent and level matrices are computed based on if the user passes in a valid pointer for them to be assigned to. Since these variables are being overwritten, at the beginning of the algorithm, lines 8 and 9, their contents are set to NULL. This is done so that if the algorithm fails, the user does not end up using bad data thinking it’s the breadth first search results.

The first 19 lines of the above code are the general set up of the algorithm. They determine which of the breadth first search values are being asked for, level, parent, or both. Lines 20 to 40 are set up for when the parent is being computed. The semiring selected for that is one that will get the node from which that node was seen, that node’s parent. It uses the ‘any’

operator in it, which is a neat operator in GraphBLAS that can be used when you need one of the values that satisfies a condition but don't care which one. This works well in this case since a node's parent can be any node in the frontier that can see it. That section also sets up the matrix to store parent information in and the frontier matrix. Then lines 41 to 53 do similar set up if the parent matrix is not being computed, setting up the frontier and semiring again. The semiring in this version only asks for a boolean, however, since for level you only need to know that the node shares an edge with a node in the frontier, not which node it shares an edge with. Lines 54 to 64 are the set-up of the level matrix if the level is being computed. This is done separately from the previous section because it needs to be done even when both level and parent matrices are being computed. Lines 65 to 92 encompass the main loop of the function. The for loop tracks two variables, the number of nodes that have been visited and the current level of the search, and stops when all the nodes have been visited or when the frontier no longer has any nodes in it. On each iteration of the loop, the frontier is advanced, checked for being empty, and provided new nodes were found for the frontier, those nodes' parent and level information are recorded as needed. The final section then sets up the parent and level matrices to be available to the user using the pointers passed in at the beginning, cleans up all created variables and returns an indication of success to the user.

2.4 Pseudo-code Estimate Diameter Algorithm

Algorithm 2: Estimate Diameter

Input: Index* diameter, Vector* peripheral, Graph G, Index maxSrcs, Index maxLoops
Output: Returns GrB_SUCCESS or an error code

```
1  // variable creation and input checking
2  Vector ecc = NULL;    // vector for tracking eccentricity
3  Vector peri = NULL;   // vector for storing peripheral node status
4  Index d = 0;          // most recently found diameter
5  Index lastd = 0;      // previously found diameter
6  Vector srcs = NULL;   // list of current sources
7  Index nsrcs;          // number of current sources
```

```
8  Matrix level              // a matrix to put level information in from the multisource bfs
9
10 bool compute_periphery  = (peripheral != NULL) ;
11 if (compute_periphery ) (*peripheral) = NULL ;
12 bool compute_diameter  = (diameter != NULL) ;
13
14 Matrix A = G->A;
15
16 Index n;
17 GrB_Matrix_nrows (&n, A) ;
18
19 // start with the first maxSrcs sources, or n if maxSrcs > n
20 if (maxSrcs > n){
21         nsrcs = n;
22 } else {
23         nsrcs = maxSrcs;
24 }
25 srcs = Vector[nsrcs];
26 for (int64_t i = 0; i < nsrcs; i++){
27         srcs[i] = i;
28 }
29
30 // core loop to find diameter
31 Monoid max = GrB_MAX_MONOID*
32 bool incSrcs = false;
33 for (int64_t i = 0; i < maxLoops; i++){
34         // save previous diameter
35         lastd = d;
36
37         // get new diameter
38         MultiSourceBFS(&level, NULL, G, srcs);
39         ecc = Vector[n]
40         GrB_reduce(ecc, NULL, NULL, max, level, GrB_DESC_T0T1);
41         GrB_reduce(&d, NULL, max, ecc, GrB_NULL);
42
43         // check for completion
44         if (d == lastd) {
45                 incSrcs = true;
46                 break;
47         }
48
49         // set up source list for the next round of the loop
50         int64_t nperi = 0;
51         for (int64_t j = 0; j < n; j++) {
52                 Index e;
53                 e = ecc[j];
```

```
54                      if (e == d) {
55                              nperi += 1;
56                      }
57              }
58              // select the number of sources for the next iteration
59              if (nperi > maxSrcs) {
60                      nsrcs = maxSrcs;
61              } else {
62                      nsrcs = nperi;
63              }
64              // choose sources
65              srcs = Vector[nsrcs];
66              int64_t curri = 0;
67              for (int64_t j = 0; j < n; j++) {
68                      Index e;
69                      e = ecc[j];
70                      if (e == d){
71                              srcs[curri] = j;
72                              curri += 1;
73                              if (curri == nsrcs) {
74                                      break;
75                              }
76                      }
77              }
78  }
79
80  // main loop complete, determine peripheral nodes if requested
81  if (compute_periphery) {
82              peri = Vector[n];
83              if (incSrcs) {
84                      for (int64_t i = 0; i < nsrcs; i++) {
85                              Index currsrc;
86                              currsrc = srcs[i];
87                              peri[currsrc] = 1;
88                      }
89              }
90              for (int64_t i = 0; i < n; i++) {
91                      Index e;
92                      e = ecc[i];
93                      if (e == d) {
94                      peri[i] = 1;
95              }
96   }
97  }
98
99  // clean up and return results
```

```
100 if (compute_periphery) (*peripheral) = peri ;
101 (*diameter ) = d ;
102 return (GrB_SUCCESS) ;
```

The first 18 lines are fairly straightforward, setting up the function. In that section, there is a check that the user has actually given a place to put the found diameter, since that is the purpose of the function. There is also a check for whether or not the user has asked for pseudo peripheral nodes to be found, since these are optional. Lines 19 to 29 set up the initial set of sources. If the user allows for more sources than there are nodes in the graph, the algorithm just gets all the nodes in the graph. Otherwise, the first maxSources nodes are selected. Lines 31 and 32 are a last bit of set up of variables. Line 31 defines the max monoid, which is just an operator that will be used later to compare a bunch of values and choose the highest one. Line 32 sets up the boolean incSrcs, a shortened form of include sources, which will be used to determine whether the source nodes should be included in the list of peripheral nodes later. I'll discuss why this is used when it is changed later.

Jumping into the core loop now, line 35 saves the previous iteration's diameter. Then the algorithm determines the current iteration's diameter. This is done in three steps. Line 38 uses the multi source breadth first search algorithm to get the level information about the sources, since the diameter should be the maximum level that can be found in that matrix. Line 40 then reduces the level matrix into the vector ecc. The descriptor in the reduce command tells the program to collapse the level matrix by column instead of by row, so instead of finding the exact eccentricity of each source node, this gets an estimated eccentricity for every node in the graph that is the furthest level it was away from any of the nodes. Finally, line 41 reduces that eccentricity vector down to the final diameter value

using max again to ensure that the diameter is the highest level that was found in the whole graph. Lines 43 to 47 then check the main end condition for the loop, which stops if the diameter found on this iteration matches the diameter found on the previous iteration. This is also where incSrcs is set to true. This is done because if the current and previous levels have the same pseudo diameter then all the sources also have that same diameter, since they were chosen as sources from the previous level for having that diameter. Lines 49 to 77 then set up for the next iteration of the loop by selecting the sources for that next iteration. It either selects all the pseudo peripheral nodes found in that iteration or the first maxSrcs pseudo peripheral nodes, if more than maxSrcs pseudo peripheral nodes were found. This concludes the loop, which will then run until either the diameter stops changing or a certain number of loops, the maximum chosen by the user, have been completed.

With the main loop complete, lines 80 to 97 find the pseudo peripheral nodes if the user asked for that. Pseudo peripheral nodes are indicated by a 1 in the peripheral node vector at their index. Lines 99 to 102 then wrap up the algorithm with the contents of the pointers passed in for outputs set to the proper values for the user and success returned.

3. RESULTS

3.1 Breadth First Search Efficiency Comparisons

Below are results from tests that were run to compare the multi source breadth first search algorithm with repeatedly performing a regular breadth first search. My initial tests were run using various graphs from LAGraph's data folder. Additional tests were done with larger graphs provided by Professor Davis. All the tests calculated both level and parent data.

Table 1: Small Graph with 34 nodes (karate.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**
1	0.000343327	0.000280804
10	0.000638823	0.00433829
20	0.000284731	0.00239744
34	0.000340607	0.00286724

Table 2: Intermediate Graph with 1000 nodes (test_FW_1000.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**
1	0.00447097	0.00360959
10	0.00744123	0.0389852
50	0.00849059	0.172488
100	0.00811515	0.341047

250	0.0167306	0.825478
500	0.029546	1.41658
750	0.035118	2.00946
1000	0.0465167	2.84693

Table 3: Large Graph with 2500 nodes (cryg2500.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**
1	0.00136034	0.00131033
50	0.0327879	0.0720413
100	0.0659519	0.144659
500	0.194916	0.730689
750	0.225545	1.08628
1000	0.312125	1.44982
1500	0.376316	2.15283
2000	0.478275	2.86541
2500	0.613305	3.69793

Note: Results showing the efficiency improvements of the multi source breadth first search algorithm over the regular breadth first search algorithm being run repeatedly. Several sizes of graph used to show the differences in improvement with different numbers of nodes, and thus iterations of the breadth first search loop.

Table 4: Push-pull Graph with 4000 nodes (pushpull.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**
1	0.0152903	0.0151979
50	0.0330606	0.74579
100	0.0431611	1.50652
500	0.142936	7.67528
1000	0.299147	15.0793
2000	54.8362	26.8286
3000	88.136	35.2629
4000	212.632	46.0929

Note: As mentioned above, I removed a section of the original breadth first search algorithm that decided between two methods of breadth first search tree expansion, push and pull. While using exclusively the push method does not usually harm my algorithm's efficiency too much, as seen by the results with the other graphs, on a graph designed to benefit from the ability to switch styles, there is a noticeable difference. *Table 4* shows the results of running the two algorithms on one such graph.

I also got the chance to test my algorithm on a couple larger, more real-world scale graphs provided by Professor Davis:

Table 5: Very Large Graph 1 with 1,134,890 nodes (com-Youtube.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**

1	0.0359982	0.0320092
10	0.25627	0.306387
50	1.77769	1.5783
100	3.16027	7.25969
500	11.3854	41.3023
1000	21.5215	80.9514
5000	103.506	462.175
10000	213.237	912.368
15000	301.597	1382.88

Table 6: Very Large Graph 2 with 1,696.415 nodes (as-Skitter.mtx).

Number of Sources	**Multi Source BFS Time (sec)**	**Repeated Single BFS Time (sec)**
1	0.0386228	0.0320368
10	0.277473	0.386897
50	2.25212	1.83416
100	4.09847	3.74342
500	16.1427	48.4041
1000	33.3384	101.595
5000	157.673	522.407
10000	325.687	1035.5
15000	510.752	1561.37

For fewer source nodes on the large graphs the single source BFS is more efficient, likely again showing the benefits of the push-pull methods. However, as the number of sources increases, the efficiency of the parallelization wins out.

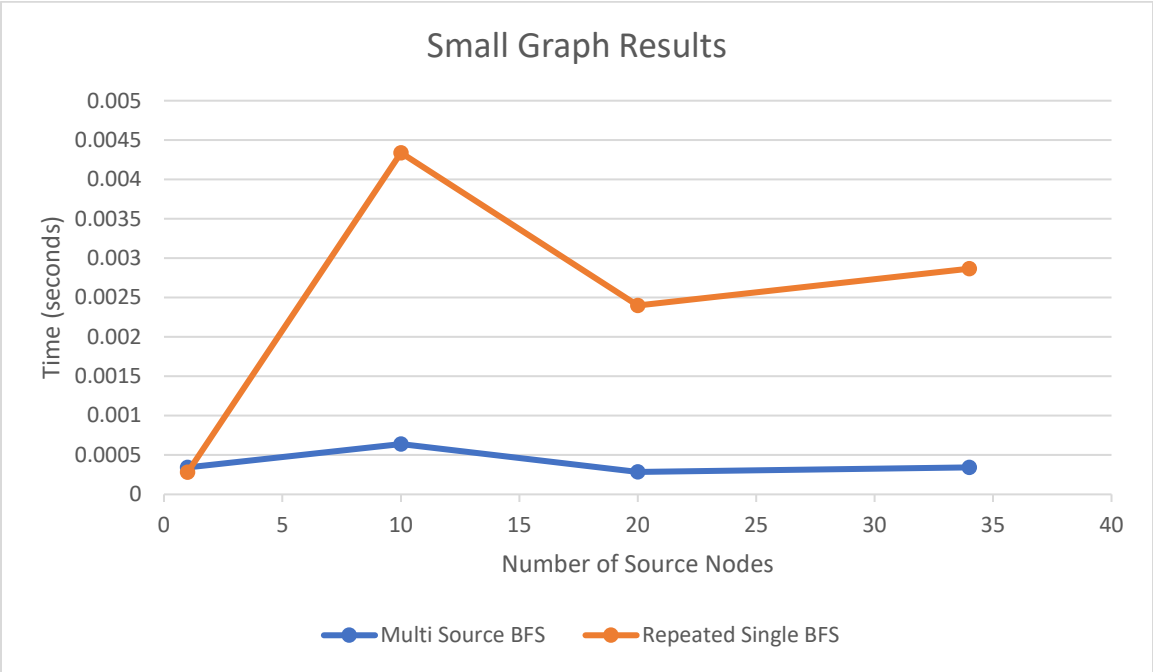

Figure 1: Graph of the results from Table 1. With so few nodes, it appears that random computational delays have significant obfuscating effects, especially in the repeated running of the single source breadth first search.

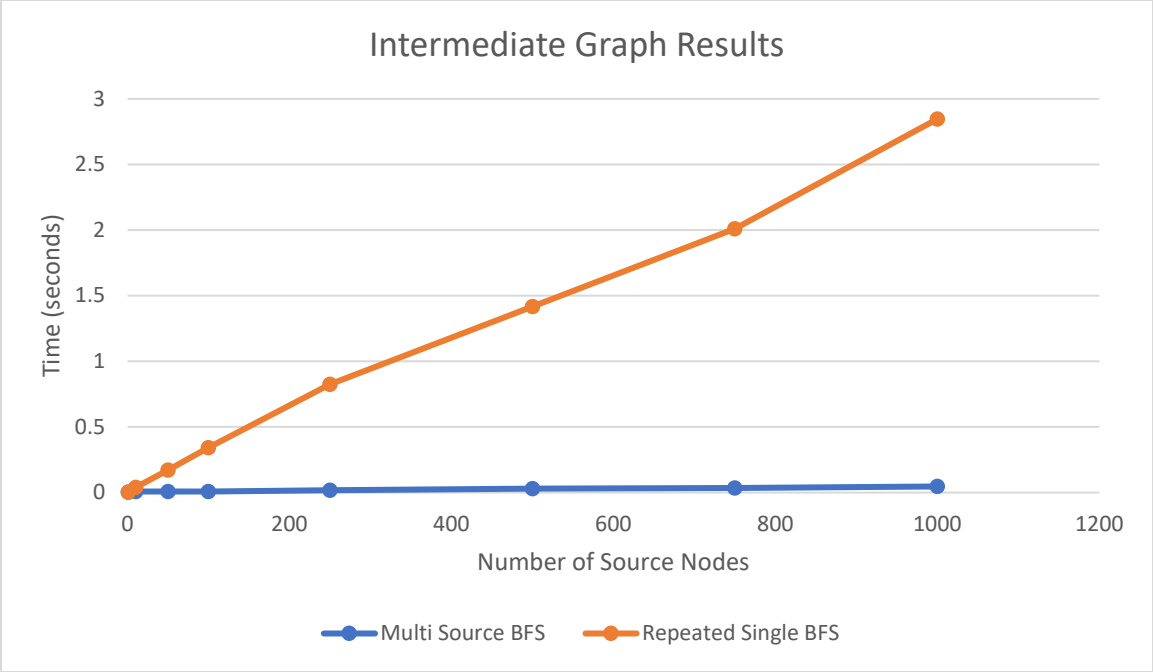

Figure 2: Graph of the results from Table 2. Time increase with the increasing number of sources begins to differ significantly, showing the benefits of parallelization.

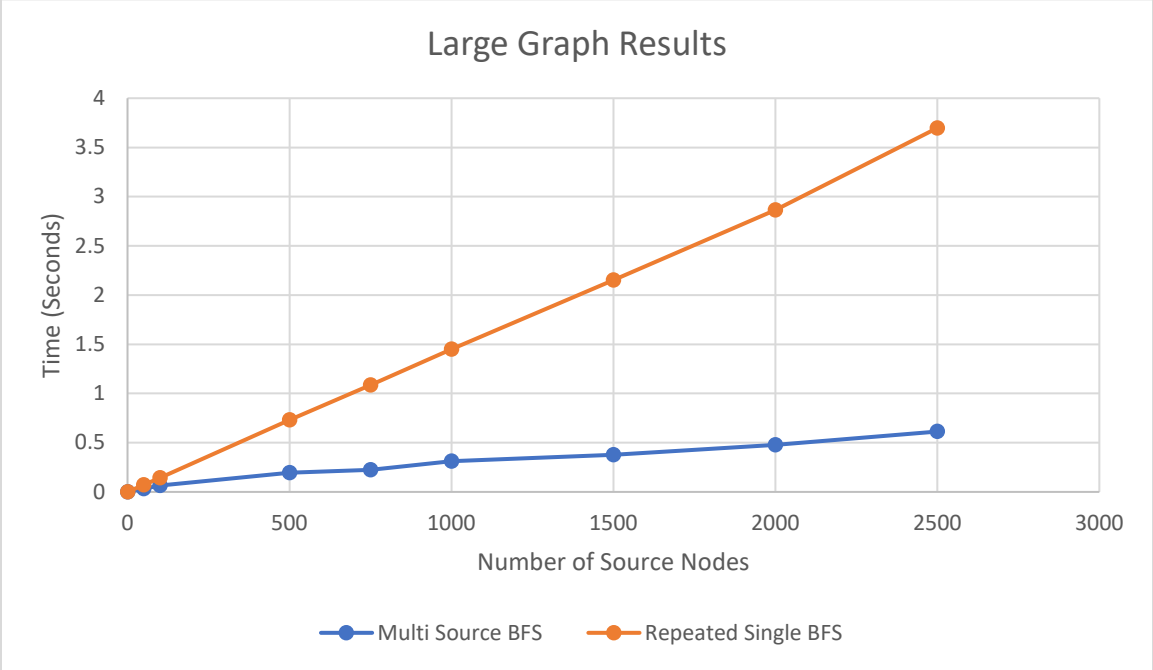

Figure 3: Graph of the results from Table 3. Increasing graph size starts to increase parallelized computation speed noticeably, but still stays better than the base algorithm.

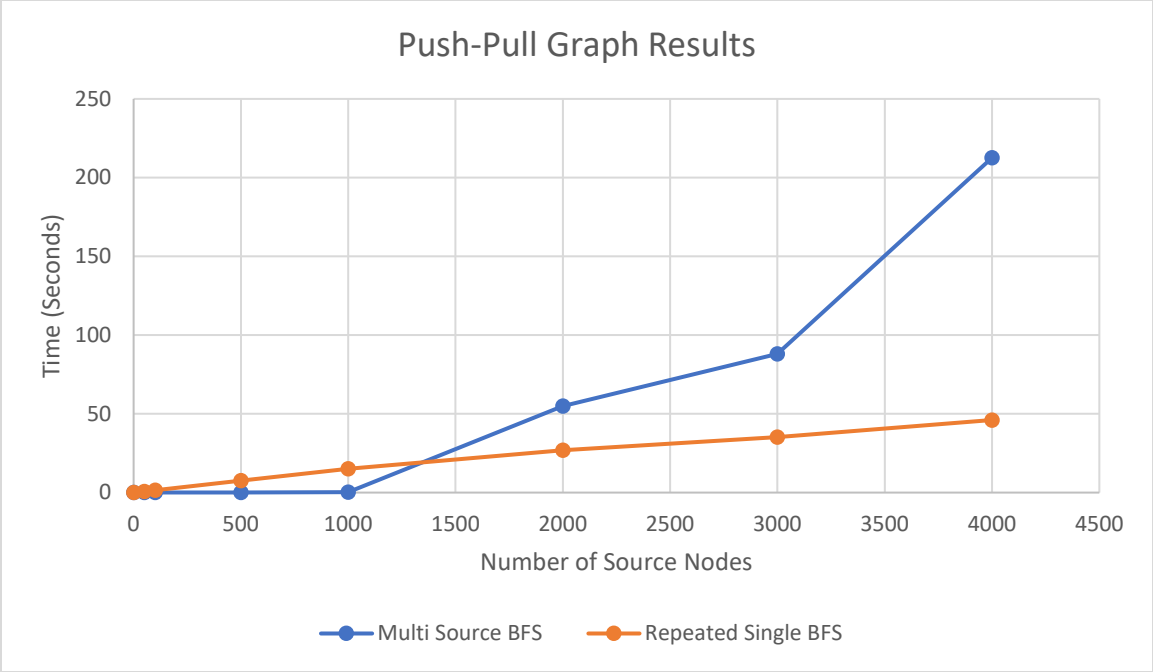

Figure 4: Graph of the results from Table 4. As discussed above, this graph demonstrates that a push-pull shift in the algorithm could provide improvements in breadth first search speed, depending on the graph.

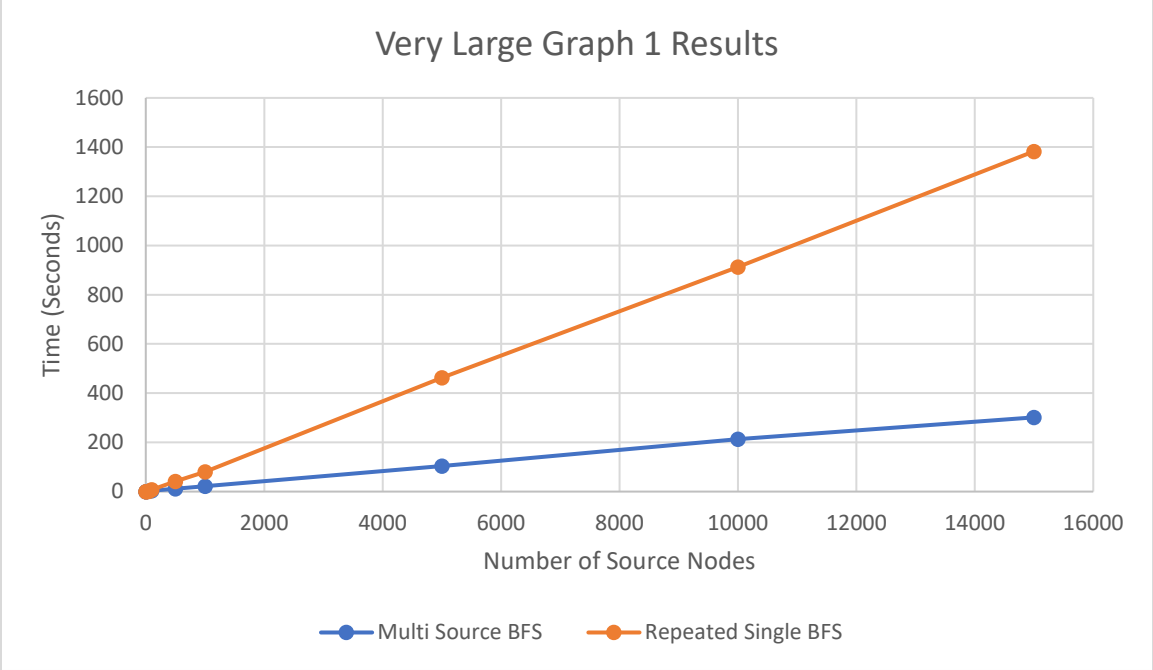

Figure 5: Graph of the results from Table 5. Both algorithms do appear to have a direct correlation between number of source nodes and the time they take.

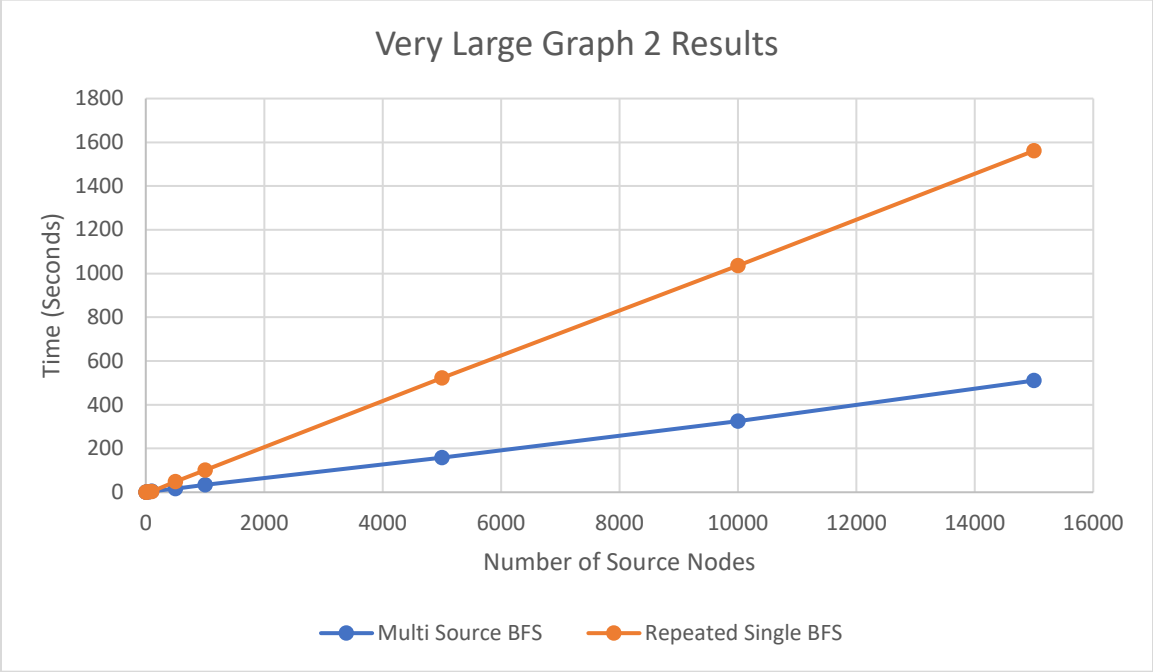

Figure 6: Graph of the results from Table 6. At this scale, the benefits of the parallelization become very apparent.

3.2 Diameter Results

Below are results from running the exact and estimated diameter algorithms. These results found with peripheral nodes being calculated. The same graphs as discussed above were used, so matching up the number of nodes can be used to determine the graph if desired.

Table 7: Diameter Results with peripheral node calculation

Graph Number of Nodes	Number of Sources per batch (k)	Exact Diameter	Exact Diameter Time (sec)	Estimated Diameter	Estimated Diameter Time (sec)
34	10	5	0.00161535	5	0.000803881
34	34	5	0.00102	5	0.00060193
1000	10	500	0.258028	500	0.0076337
1000	100	500	0.051773	500	0.0133934

1000	500	500	0.0402269	500	0.0259454
2500	10	98	1.83667	98	121.036
2500	100	98	1.67881	98	165.649
2500	1000	98	0.700281	98	168.409
2500	2500	98	0.498017	97	167.978
4000	100	2005	12.3621	2005	0.0510473
4000	500	2005	53.3279	2005	0.1686
4000	2000	2005	132.765	2005	55.6341
4000	4000	2005	208.09	2005	214.463
1,134,890	10	X	X	24	0.523146
1,134,890	100	X	X	24	2.01806
1,134,890	500	X	X	24	7.01838
1,134,890	1000	X	X	24	13.2149
1,134,890	5000	X	X	24	61.771
1,696,415	10	X	X	31	98.3638
1,696,415	100	X	X	31	2.7098
1,696,415	500	X	X	31	9.62034
1,696,415	1000	X	X	31	20.4759
1,696,415	5000	X	X	31	98.189

4. CONCLUSION

4.1 Discussion of Multi Source Breadth First Search Results

For small graphs of less than 100 nodes, the speed up from this algorithm is unlikely to be noticeable. The convenience of not having to write a loop to go through the source nodes would probably be of more benefit to the programmer. As the graphs reach a few thousand nodes, the benefits of the algorithm's parallelization become noticeable. At this size we do still see very reasonable speeds from both algorithms and some graphs which particularly benefit from switching frontier advancement methods mid-way through are even more efficient with repeated use of the single source algorithm. Finally, for real world scale graphs, graphs with over a million nodes, the number of source nodes plays a role in which algorithm is most efficient. With as many as 100 source nodes, the benefits of the single source algorithm like the push-pull advancement can be more efficient than the multi source breadth first search algorithm. However, once one reaches up to 500 or more source nodes, the advantages of the algorithm's parallelization outweigh these benefits. I suspect that the reason the single source algorithm performs well for so long is that the benefits of the ability to change between push and pull methods becomes more pronounced as the size of the graph increases. The graphs of the real world scale tests of the algorithm also show that there seems to be a linear correlation between the number of source nodes and the time it takes the algorithms to run, though this relationship was less apparent on smaller graphs.

4.2 Discussion of Exact and Estimated Diameter Results

My main expectations for the exact diameter finding algorithm were that adding more nodes to the number of sources per batch would speed it up since it would allow for greater

parallelization and that it would in most cases be slower but more accurate than the estimated diameter. For the estimated diameter, I expected that it would be fairly quick but not always accurate and that increasing the number of sources per batch would usually make it slower but might allow it to be more accurate since it could explore nodes it wouldn't find on smaller iterations. While most of these expectations were validated by the results, there were a few interesting deviations. The oddest was the 2500 node graph, which I suspect had some oddity in how it was laid out given the difference in behavior between it and all the other graphs. Especially odd was the last iteration, where it should have used all 2500 nodes as sources and thus found the exact diameter immediately and confirmed it on the second iteration. Instead, however, that was the only iteration to find a lower diameter than others on a given graph. I am currently uncertain what caused this behavior.

On the very large graphs, I was unable to compute the exact diameters due to computing resource constraints, but I was able to get some interesting results from the diameter estimation algorithm. In particular, the time difference on the largest graph between 10 source nodes and 100, which I hypothesize is due to the 100 source nodes then including source nodes that can reach the diameter in far fewer iterations.

4.3 Conclusion

In conclusion, this algorithm allows a user of LAGraph to more efficiently gain BFS information from several source nodes in a graph. In most cases, the multi source breadth first search algorithm can be used to obtain breadth first search data about several source nodes more efficiently than repeatedly using the normal breadth first search algorithm. However, some exceptions can occur when a graph greatly benefits from a pull method of frontier advancement later in the algorithm or when finding relatively few BFS trees on a very large graph. As such,

one way this research could be further improved in the future would be investigating ways to add the push pull functionality to the multi source BFS algorithm. In particular, determining a good time to switch methods of frontier advancement would likely be the most interesting continuation on that path.

Beyond its direct value to users, the multi source breadth first search algorithm can be used to determine other interesting information. I explored calculating the diameter of a graph, which would be the highest level found if a BFS were performed on every node in the graph. Additionally, rather than performing the BFS on every node, the diameter can be estimated by selecting only a sample of the nodes and by iterating on a sample of the nodes that exhibit the highest BFS level until the highest level found no longer increases. Finding the exact diameter can also find peripheral nodes of the graph, since they are the ones seeing that diameter and in the same way pseudo-peripheral nodes can be found when estimating the diameter.

In the future, the BFS data could also be used to estimate a good cut of the graph by cutting all the nodes at a certain level of the graph away from a peripheral or pseudo-peripheral node, such as making a cut at a level near half the diameter of the graph. While such a cut is unlikely to be completely optimized, it could aid other algorithms in running more efficiently by giving them a reasonable starting point rather than having them work from a random cut. Some other information about a graph that could be found or estimated using similar methods are a graph's center, eccentricity, and radius.

REFERENCES

[1] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, “Exploring network structure, dynamics, and function using NetworkX”, in *Proceedings of the 7th Python in Science Conference* (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[2] Weisstein, Eric W. "Graph Diameter." From *MathWorld--A Wolfram Web Resource*. https://mathworld.wolfram.com/GraphDiameter.html

[3] T. Davis, “GraphBLAS,” Graphblas. [Online]. Available: https://people.engr.tamu.edu/davis/GraphBLAS.html. [Accessed Sept. 12, 2022].

[4] J. Antal et al. (2023) LAGraph (v1.0.1) [Source code]. https://github.com/GraphBLAS/LAGraph. [Accessed Oct. 5, 2022]

[5] T. Davis, (2022) LG_BreadthFirstSearch_SSGrB [Source code]. https://github.com/GraphBLAS/LAGraph/blob/stable/src/algorithm/LG_BreadthFirstSearch_SSGrB.c. [Accessed Oct. 25, 2022]