MATCHING AND COARSENING IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

VIDITH MADHU

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the 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, Vidith Madhu, 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

Page

ABSTRACT

Matching and Coarsening in GraphBLAS

Vidith Madhu
Department of Computer Science and Engineering
Texas A&M University

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

Recently, there has been a significant desire both within the scientific community and industry to write graph algorithms using linear algebraic operations. This leads to algorithms that can leverage many important algebraic properties of matrix operations, as well as the vast body of research conducted in high performance and parallel linear algebra computations. In addition, such formulations usually lead to highly expressive and short code. To this end, SuiteSparse:GraphBLAS is a framework developed to easily write graph algorithms in the language of linear algebra. LAGraph is a test harness and collection of algorithms written with SuiteSparse; this work will detail the contribution of new algorithms to this collection, which perform maximal matching and coarsening of undirected graphs.

A matching is a subset of the edges of a graph such that no two edges in the set share a common vertex. A maximal matching is one that is not a proper subset of any other matching. Finding any maximal matching is a simple process, but it is hard to find ones with optimal characteristics (i.e. maximum sum of edge weights, maximum size of matching, etc). Coarsening refers to reducing the size of a graph in a manner that preserves connectivity information. One way

to achieve this is by collapsing edges in a maximal matching. It is known from prior work that matching-based coarsening produces small graphs that can be easily bisected (a process known as multilevel bisection). By projecting the coarsened graph back to its original size, and applying refinements at each stage, multilevel bisection can be recursively used to approximate good k-way partitions, which is known to be an NP-complete problem.

We begin with a discussion of the mathematical background behind linear algebraic graph algorithms and the GraphBLAS standard. We then discuss graph partitioning and multilevel techniques further in depth. Finally, we present our algorithms, their performance results, and discuss avenues for future work.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Tim 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.

Finally, thanks to my parents for their encouragement and love.

The image used for Figure 2 is from Wikimedia Commons and used under the Creative Commons Attribution-ShareAlike 3.0 Unported License, which can be found here.

Funding Sources

Undergraduate research was supported by Dr. Timothy Davis at Texas A&M University.

1. INTRODUCTION

1.1 Graph Algorithms in the Language of Linear Algebra

There has been a long observed duality between graphs in their standard representation as a set of vertices and edges and representations in matrix form (such as an adjacency matrix). In fact, this connection was noted alongside the inception of graph theory itself by Konig. In recent years, there has been a mounting desire in both academic and industrial sectors to develop graph algorithms using matrix operations. Not only are such algorithms concise and highly expressive, they can also leverage the large body of research conducted in high performance linear algebra computations and exploit properties of matrix operations to yield better performance than their conventional counterparts.

1.1.1 GraphBLAS

GraphBLAS [1, 2] is a standard for developing graph algorithms using matrix operations that has gained great popularity in recent years. Like many other linear algebra standards, it is influenced by the BLAS (Basic Linear Algebra Subprograms) specification. It utilizes several algebraic properties of matrix multiplication to allow for the development of a wide variety of graph algorithms using a relatively small feature set. The major pieces that constitute GraphBLAS are collections (such as matrices and vectors, which may contain entries of a variety of data types), algebraic objects (such as semirings, monoids, and binary/unary operators), and operations (such as `mxv` (matrix-vector multiply) and `eWiseMult` (element-wise multiply)). One of the key properties that GraphBLAS takes advantage of is the consistency of matrix multiplication over an algebra of semirings.

A semiring is an algebraic structure (D, $\oplus$, $\otimes$), consisting of a set D and two binary operators, one additive ($\oplus$), and one multiplicative ($\otimes$), both of which take $D \times D \rightarrow D$. The additive operator must have an identity (the zero element) which annihilates multiplication (meaning for all $x \in D$, $x \otimes 0 = 0$). We can further describe the additive operator as a monoid (D, $\otimes$). A monoid is

another algebraic structure that generalizes the notion of a group; for the purposes of GraphBLAS algorithms, the monoid can be thought of as the part of the semiring that joins the results of individual multiplications into a single scalar. There are a handful of important differences between the rigorous definition of a semiring and that used within GraphBLAS. For example, in GraphBLAS, $\otimes$ need not distribute over $\oplus$. Such relaxations greatly increase the flexibility of algorithm development and result in better performance, while potentially sacrificing some conventional properties of matrix multiplication. Additionally, GraphBLAS semirings admit two input domains D_{in1} and D_{in2} and a codomain D_{out}, which is useful when there is a mismatch in the data types of two input structures that can be reconciled via casting. By this definition, $\otimes : D_{in1} \times D_{in2} \rightarrow D_{out}$ and $\oplus : D_{out} \times D_{out} \rightarrow D_{out}$.

Intuitively, the purpose of semirings in GraphBLAS is to control the behavior of matrix multiplications. In canonical matrix multiplication, the semiring used is `PLUS_TIMES`; the monoid, which is listed first, is `PLUS`, which is arithmetic addition, and the multiplicative operator is `TIMES`, which is artihmetic multiplication. If we perform a multiplication $A\vec{b}$ over `PLUS_TIMES`, we can think of first performing an element-wise multiply on each row of A with $\vec{b}$ using `TIMES` to get a new matrix A'. Then, we reduce the rows of A' to a single scalar using `PLUS` to obtain the result. If, for example, we changed the monoid to `MIN`, the reduction would take the minimum of each row of A' instead of the sum of entries in the row. One important point that must be understood to clear potential confusion is the role of the implied zero in GraphBLAS. When the semiring applied in a matrix multiplication has an additive identity that annihilates multiplication, we can take advantage of sparsity by having non-existent entries for any zero element, and only consider the results of multiplications where both entries are present. This works because multiplications with zero will not contribute anything to the summation by the monoid. In other words, we can use a sparse representation of the matrix (storing values and positions of entries) rather than a dense one (storing an explicit structure that matches the shape of the matrix). All of the semirings presented in this work take advantage of this implied zero feature.

1.1.2 SuiteSparse and LAGraph

SuiteSparse:GraphBLAS [3] (herein referred to as SuiteSparse), is the first implementation of the GraphBLAS standard that is fully compliant, and acts as the official reference implementation of GraphBLAS. It was developed by Timothy A. Davis at Texas A&M University, and is used to develop the algorithms detailed in this work. LAGraph is a collection and test harness for algorithms developed using SuiteSparse; we will refer to some algorithms in LAGraph that influenced the development of the algorithms in this work.

1.1.3 Breadth-First Search in SuiteSparse

To demonstrate the application of SuiteSparse in a real algorithm, we will walk through a basic implementation of breadth-first search of an unweighted graph. We need to have the $N \times N$ Boolean adjacency matrix A and a Boolean vector $\vec{v_0}$ of size N. $\vec{v_0}$ will mark the source vertex, and all other entries will be empty (implied zeroes). To perform one step of the BFS, we can do $\vec{v_{i+1}} = A^T\vec{v_i}$ over `ANY_ONE`[1]. `ANY` is an example of a monoid that has been introduced in SuiteSparse but not specified in GraphBLAS. In particular, `ANY` is special in that it is non-deterministic: `ANY(X, Y)` can result in either `X` or `Y`. As such, using this monoid when possible can lead to performance optimizations at runtime, when we simply want the monoid to output the result of any multiplication. The `ONE` operator is simply defined as `ONE(X, Y) = 1`. For example, if we did $\vec{v_1} = A^T\vec{v_0}$, $\vec{v_1}$ will mark all the entries that are one step away from the source vertex marked in $\vec{v_0}$. The intuition for how this works should be clear if we revisit the explanation of the `PLUS_TIMES` semiring in section 1.1.1 and substitute it with the `ANY` monoid and `ONE` operator. While relatively simple, there is a key advantage that this formulation has over a standard BFS implementation, in that it is inherently parallel. Thus, this simple example demonstrates one of they key reasons to formulate graph algorithms in this manner.

[1] In SuiteSparse, the semirings are named as follows: `G<x/r>B_<add>_<mult>_<type>`. The prefix `GrB` is used for semirings defined in the GraphBLAS standard, whereas `GxB` is used for those exclusively in SuiteSparse. `<type>` refers to the data type of the first argument of the multiply op. So, in this case, the full name would be `GxB_ANY_ONE_BOOL`.

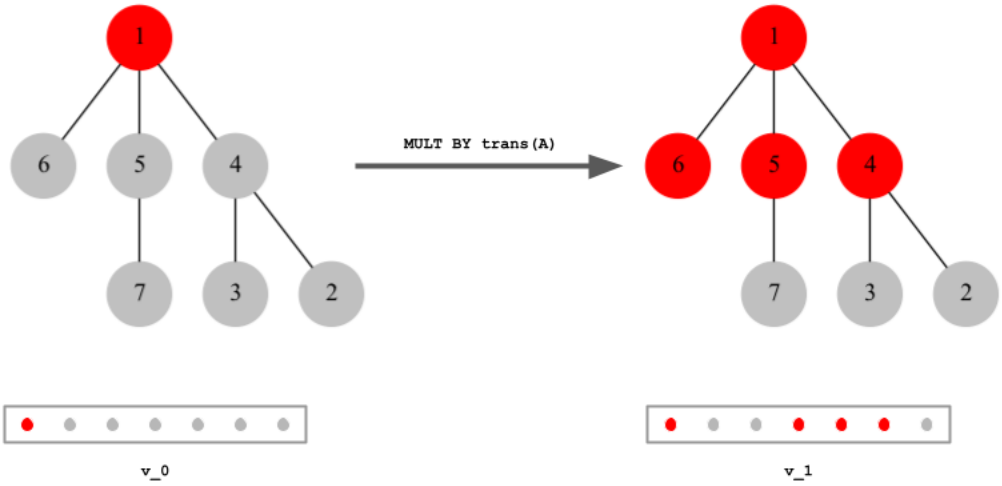

Figure 1: Illustration of a step in a basic BFS with GraphBLAS

1.2 Graph Partitioning

Graph partitioning refers to the problem of assigning labels to the vertices of a graph such that the total weight of edges incident to vertices of different labels (called the *edge-cut* of the partition) is minimized. Formally, if we have a graph $G = (V, E)$, a partition is a set of pairwise disjoint subsets $\{V_1, V_2, ...V_k\}$ where $\bigcup_{i=1}^{k} V_i = V$. The focus of this work is on methods that build towards k-way graph partitioning, where we enforce that we must partition the graph into exactly k subsets of roughly equal size. We also focus only on undirected graphs with no self-loops. This problem is known from prior work to be NP-complete, but has numerous real-world applications[2] [4]. For example, we can model a distributed system using a graph, where the vertices are tasks that must be performed, and edges encode overhead costs incurred by communication between tasks. The problem of allocating these tasks to k compute nodes while minimizing communication

[2]This problem is closely related to the minimum cut problem. In general, finding the *k-min cut* can be done in polynomial time if k is fixed. However, this is no longer the case if we require a balanced partition; this is what differentiates k-way partitioning from min cut.

costs and achieving good load balancing reduces to k-way graph partitioning. Numerous methods have been developed to approximate k-way partitions; this work builds towards a technique known as multilevel bisection, which uses maximal matching and coarsening (the focus of the algorithms presented in this work) as subroutines.

1.2.1 Maximal Matching

Given a graph $G = (V, E)$, a matching $E' \subset E$ is a subset of edges such that no vertex in the graph is incident to more than one edge in the matching. A matching is maximal if it is not a proper subset of any other matching. In particular, it is important to understand the difference between *maximal* and *maximum*. A maximum matching is a maximal matching that is maximized by some metric, such as the size of matching or the sum of edge weights in the matching. It is not necessary for a maximal matching to be maximum since a matching can be chosen in a non-optimal manner. In fact, finding any maximal matching is relatively easy; we can show that greedily choosing edges in some arbitrary order will result in a maximal matching. However, for many real-world use cases, it is useful to find maximal matchings that are close to maximum. For example, HEM (heavy edge matching) is a heuristic used in multilevel bisection where the total weight of the matching is maximized [5]. Doing so produces partitions with smaller edge cuts.

1.2.2 Multilevel Bisection

A common approach to partitioning a graph is to recursively bisect it. A bisection is a partition of a graph into two subsets of roughly equal size; we can then bisect each subset as needed until we reach k partitions. An iterative multilevel technique can be used to reduce the graph so that it can be easily bisected by heuristics such as the Kernighan-Lin algorithm (KLA) [6]. This reduction is done by first finding a maximal matching as described above, and then applying a coarsening operation on the graph with that matching. In this context, coarsening refers to collapsing matched edges, and merging the endpoints of each matched edge into a vertex. In particular, we would like to consider these new vertices as a cluster of original vertices in the graph. It is also possible that edges that form a triangle with a matched edge will collapse into a

single edge. In such cases, it is typical to sum the edge weights so that we encode the total weight of edges incident to one of the vertices in the cluster. Once the coarsened graph is bisected, it must be projected back to its initial size. Given the sequence of coarsened graphs $G_1, G_2, ..., G_n$ with $G_i = (V_i, E_i)$ (G_1 being the original graph), let $P_i : V_i \rightarrow \{0, 1\}$ map vertices in G_i to a bisection, and for all $v \in V_i$, let $C_v \subset V_{i-1}$ be the vertices of V_{i-1} contained in v. To obtain P_{i-1}, we can simply do $\forall v \in V$ and $\forall u \in C_v, P_{i-1}(u) = P_i(v)$. However, this approach can upset the quality of the bisection as the uncoarsening proceeds. To address this, local refinement strategies are employed that improve a given P_i to be more optimal (a popular choice is again the Kernighan-Lin algorithm). There exist other multilevel techniques that are used to indirectly partition graphs. For example, a multilevel approach may be used to speed up the computation of the Fiedler vector of a graph, which allows for a technique known as spectral partitioning. However, this is a separate topic and is not related to this work.

2. METHODS

2.1 Maximal Matching

2.1.1 Luby's Algorithm

Luby's algorithm [7] is one of the well-known, practical solutions for finding maximal independent sets of graphs. Briefly, a maximal independent set of an undirected graph is a maximal subset of vertices of the graph such that no two chosen vertices are adjacent to each other. The algorithm works by performing several rounds; in each round, the remaining vertices in the graph are assigned random scores weighed by their degrees (lower degrees are given higher scores). Then, vertices which have larger scores than all of their neighbors are included in the result. The chosen vertices and their neighbors are eliminated from the graph, and the next round begins[1]. What distinguishes Luby's method from other, naive MIS algorithms is that the core components of the algorithm, the score assignment and vertex selection, can be done in parallel. This also means it is highly suitable for implementation in SuiteSparse. In fact, there exists an experimental LAGraph function `LAGraph_MaximalIndependentSet` that uses a modified version of Luby's method. While not an explicit approach for maximal matching, it turns out that we can use the same principles of Luby's method to build our maximal matching implementation.

2.1.2 Line Graphs

In order to see how to apply Luby's method to the maximal matching problem, we can transform the maximal matching problem to a maximal independent set problem. In particular, this means that given a graph $G = (V, E)$, we want to somehow transform it into $G' = (V', E')$, where there is a bijection $f : V' \to E$ and there exists $(u', v') \in E'$ if and only if $f(u')$ and $f(v')$ share a vertex in G. From an intuitive sense, we are building a new graph that represents the adjacency between edges in the original graph. Such a graph is called a *line graph*.

[1] It is easy to see that this algorithm is correct. Let u and v be neighboring vertices, and $f(u)$ and $f(v)$ be their respective scores. For both vertices to be added to the set, they must both be chosen in the same round. This forms a contradiction, since it implies $f(u) > f(v)$ and $f(u) < f(v)$.

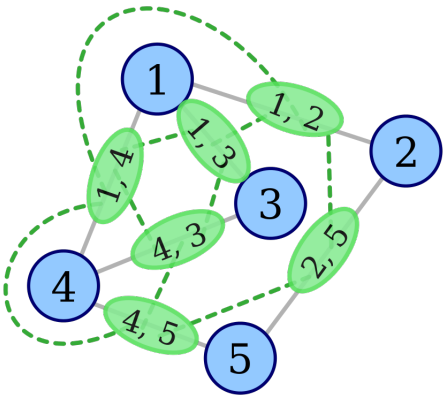

Figure 2: Illustration of a line graph. Line graph vertices are green ovals, and edges are dashed green lines.

2.1.3 Maximal Matching in SuiteSparse

We use the `LAGraph_MaximalIndependentSet` function mentioned earlier as a basis for our implementation. Given a graph with N vertices and M edges, the MIS function takes in the $N \times N$ adjacency matrix A of the graph as input. So, we want to find a method to transform A to A', where A' is the adjacency matrix of the line graph. To do this, we introduce a new matrix encoding of a graph, called the *incidence matrix*. The incidence matrix E is a $N \times M$ matrix where $E_{ij} \neq 0$ if and only if vertex i is an endpoint of edge j. As is the case with adjacency matrices, the value of the non-zero entries themselves can be of any type, depending on if we want to encode edge weights or simply the connectivity pattern of the graph. As we will see later, for our algorithms we want E to contain edges weights. However, for now, it suffices to have E' be a Boolean matrix, meaning A' will also be Boolean. In addition, LAGraph (the algorithm collection we are contributing to) offers a wrapper object for graphs called `LAGraph_Graph`, which contains the adjacency matrix A of the graph. So, given some black box function `A_to_E` which can convert A to E, we can concisely express the desired translation as follows:

$$A' = E^T E - I \tag{1}$$

where I is the $M \times M$ identity matrix. In this context, subtraction means eliminating any non-zero entries in the result that line up with non-zero entires in the right-hand side matrix. In GraphBLAS, this can be done either explicitly with `eWiseMult`, or can be achieved by using I as a complemented mask when performing the matrix multiplication. This subtraction is needed as performing $E^T E$ introduces self-loops from each edge to itself. We can use `ANY_ONE` as our semiring for the matrix multiplications. However, there is a significant problem with this formulation that must be addressed, which is memory consumption. In particular, explicitly computing A' for certain graphs can result in memory blow-ups. Specifically, in order to store a matrix with k entries, it is necessary to use at least $O(k)$ space. Let us consider the basic example of a star graph, where there is a central "hub" vertex, and edges only from the hub to every other vertex in the graph. Let the number of edges in this graph be M; hence, we need $O(M)$ space to store A. It is clear that the line graph will have M^2 edges, meaning A' will consume $O(M^2)$ space. Even with a modest M such as 10^5, using the above formulation is infeasible on most workstations. Therefore, we must seek an implicit approach to using A'. In other words, we want to perform all operations that use A' without explicitly computing it using the above formula.

Assume we have A'. One of the steps we need to perform according to Luby's method is computing the maximum score neighboring edge of every edge. Given some vector $\vec{x}$ that contains the scores of all edges currently in the graph, computing $A'\vec{x}$ over `MAX_SECOND` would produce the desired result. We can use Equation 1 to expand this to $(E^T E - I)\vec{x}$. What we ideally want is to be able to distribute this as $E^T E\vec{x} - I\vec{x}$, and then use associativity to compute $E^T E\vec{x}$ as $E^T(E\vec{x})$. This would be possible with the conventional `PLUS_TIMES` semiring, however `MAX_SECOND` does not behave nicely in this sense, since there is no additive inverse for `MAX`, and hence the subtraction cannot be performed. So, we must discard this exact representation of A', and instead redefine A' to be the adjacency matrix of the line graph with self-loops, allowing us to still use associativity and hence use A' implicitly:

$$A' = E^T E \tag{2}$$

One of the immediate concerns that arises from using A' in an implicit manner is the effect of having self-loops. Indeed, we must reinterpret the vertex selection step of Luby's algorithm, since the max score neighbor of a particular edge will now include the score of that edge itself. This means that it will be impossible for the algorithm to make progress, since no edge can have a score greater than its own. For our implementation, we change the condition for including an edge in the matching to check for equality instead of a strictly greater score. This gives rise to another concern, which is that adjacent edges sharing the same score will both be included in the matching. To address this, we simply repeat the vertex selection if such an invalid matching is produced. While this may initially seem massively detrimental to performance, as we see in the results our algorithm still performs quite well. In particular, it is highly unlikely for the vertex selection step to produce an invalid matching given a wide enough range of possible scores.

Algorithm 1 Maximal Matching in SuiteSparse

Input

E incidence matrix
RE pointer for result
MT matching type (RM, HEM, LEM)

Output

None

1. Initialize all objects (E^T, misc. `GrB_Vector` and `GrB_Matrix`)
2. Use "2-hop" step to compute edge degree. Use E to build weight vector
3. Generate random seed for each edge. Add all edges to candidate set
4. While candidate edges remain:
   (a) Generate scores by weighing seeds by edge degree
   (b) Based on MT (matching type), further weigh scores by edge weight
   (c) Perform "2-hop" step to get max score neighbor for each edge
   (d) Select edges that have score $\geq$ max neighbor score
   (e) Verify selected edges form a matching. If not, reseed and return to (a)
   (f) Perform "2-hop" step to get selected edges + their neighbors
   (g) Remove all edges found in previous step from candidate set, add to result
   (h) Reseed and return to (4)
5. Set RE = result, free allocated objects

The core computation that occurs throughout the algorithm as a result of using A' implicitly is a process that can be intuitively thought of as a sequence of two "hops": One from edges to vertices, and the next from vertices back to edges. Let $\vec{v} = E\vec{x}$ for some vector $\vec{x}$. Then, the first hop computes $\vec{v}$, and the second hop computes $E^T\vec{v}$. This process is concretely implemented by two successive `GrB_mxv` operations. Assuming our graph has N vertices and M edges, $\vec{x}$ will correspond to an "edge" vector of length M, $\vec{v}$ will correspond to a "vertex" vector of length N, and the result $E^T\vec{v}$ will again be an edge vector. Tying both hops together yields a single BFS-like step in the line graph. It then becomes clear how the implicit computations with A' correspond to a single matrix-vector multiply using an explicitly built A'; this makes it relatively straightforward to interpret the above algorithm in terms of the steps of Luby's method.

It should also be noted that we provide the user with multiple options for how they want to optimize the matching towards a maximum metric. The default is random matching (RM), which uses the same approach as `LAGraph_MaximalIndependentSet`. However, for the purposes of coarsening, it may be beneficial to generate heavy-edge matchings (HEM) or light-edge matchings (LEM) as discussed earlier, as greedy heuristics that minimize the cost of the bisection of the final coarsened graph. For this reason, we also build a weight vector in our implementation, and is the reason we want E to contain edge weights (we can use `GrB_reduce` on E to build a weight vector). For weighted matchings, we perform an arithmetic element-wise multiply between the weight vector and scores to favor heavier or lighter edges.

Note that in the above algorithm, we do not use the black box `A_to_E` function as discussed earlier. This is because the coarsening function also needs access to the incidence matrix E, so in the context of this work it is more efficient to compute the E matrix in each coarsening step and pass it to the maximal matching function. We will discuss the implementation of the `A_to_E` function in section 2.3.

2.2 Coarsening

Generally speaking, coarsening from an undirected graph G_i to G_{i+1} where $G_i = (V, E)$ and $G_{i+1} = (V', E')$ involves choosing an bijection $f : V \to V'$ where for all $v \in V$ if $f(v) \neq v$, then

$f(f(v)) = f(v)$. Then, we construct G' such that for all $(u, v) \in E$, we add $(f(u), f(v))$ to E' if and only if $f(u) \neq f(v)$. Oftentimes, we want to know the strength of connection between vertices in the coarsened graph. If so, we can slightly alter the procedure for coarsening to instead have an inserted edge $(u', v') \in E'$ have an edge weight that is the sum of weights of edges $(u, v) \in E$ such that $f(u) = u'$ and $f(v) = v'$. There are numerous methods to coarsen a graph. For example, in directed graphs, a meaningful choice of f could be based on which strongly connected component a vertex belongs to. In our case, we want to generate the mapping f from a maximal matching, which will intuitively result in a collapsing of edges. Fortunately for us, there exists an established method for coarsening graphs in parallel if we know what f is, meaning it can be directly implemented in SuiteSparse. This involves building a Boolean S matrix from f, where $S_{ij} = 1$ if and only if $f(j) = i$. So, columns in this matrix correspond to the uncoarsened (original) vertices, and rows correspond to coarsened vertices. To perform a single coarsening step given the adjacency matrix A, we can do $A' = SAS^T$, where A' corresponds to the coarsened adjacency matrix. The choice of semiring depends on whether we want the edge weights to sum as described above. If so, we can use `PLUS_TIMES`, else `ANY_ONE`. The shape of this S matrix depends on how we want to do the coarsening: When applying the coarsening, vertices where $u \neq f(u)$ will be removed from the graph. However, this removal can be interpreted in different ways. If we want to retain the size of the graph (for example, to easily identify which vertices in the original graph correspond to vertices in the coarsened graphs), we can simply "exile" these nodes by making them singletons[2] for the remainder of the coarsening scheme. Alternatively, we can compress the size of the graph and explicitly remove vertices that will disappear once a coarsening is performed, meaning we will need to potentially relabel many of the vertices in the graph. This will perform better for size-sensitive algorithms that use the coarsening results, but will require maintaining mappings of old vertices to new vertices for each coarsening step.

[2]A singleton is a vertex with degree 0.

Algorithm 2 Coarsening in SuiteSparse

Input

G LAGraph graph object (contains A)
REA pointer for result adj mat
REM pointer for result parent mappings
MT matching type (RM, HEM, LEM)
PW Whether to preserve graph size
CW Whether to combine edge weights
N Number of coarsening levels

Output

None

1. Initialize all objects (E^T, result A, misc. `GrB_Vector` and `GrB_Matrix`)
2. While $N > 0$:
   (a) Use `A_to_E` to convert the current A matrix to a E matrix
   (b) Rebuild E^T using current E
   (c) Run maximal matching function with E and MT
   (d) Build S matrix (details in 2.3.2)
   (e) Update REM with generated parent mapping, if PW is false (details in 2.3.2)
   (f) Run SAS^T to get the next A matrix, using CW. Decrement N
   (g) Free objects allocated for this coarsening step
3. Set REA, REM to appropriate results, free allocated objects

2.3 Utility Methods

2.3.1 Building Incidence Matrices

As referenced in the above sections, we need an `A_to_E` function in order to build the required incidence matrix for our algorithms to operate on, due to it not being already available as an attribute of `LAGraph_Graph`. To do this, we note that entries in an adjacency matrix A correspond to edges in the graph. Therefore, we need a method to extract all non-zero entries in A and build an incidence matrix E with them. The GraphBLAS standard defines such functions: `GrB_extractTuples` and `GrB_Matrix_build`. In particular, `GrB_extractTuples` gives us the rows, columns, and values of non-zero entries via user-provided arrays. It is important to note that since we are dealing with undirected graphs, A will be symmetric, so to ignore redundant entries we only consider upper-triangular entries. Suppose the i-th upper-triangular entry is given as a tuple (r, c, v) corresponding to a row, column, value in A. We want to create two entries in E in the i-th column for the two endpoints of this edge. Specifically, this will be $E_{ri} = v$, and $E_{ci} = v$.

2.3.2 Converting Matchings to Mappings

In our coarsening method, we want a function that will allow us to convert the result produced by our maximal matching function into the S matrix as discussed above to perform coarsening. For this, we want a two step process: Note that if our graph has N vertices and M edges, the matching vector is a Boolean vector of length M marking which edges are selected as part of the matching. The first step involves translating this vector to a new vector $\vec{p}$ of length N that specifies a parent mapping on the vertices of the graph. In particular, referencing the definition of a coarsening at the beginning of section 2.2, $\vec{p}_i = f(i)$. The key property to observe here is that every vertex will be incident to at most one matched edge. So, for a matched edge, we arbitrarily choose one of the endpoints to be the parent of both endpoint vertices. To implement this, we can first perform a multiplication with E^T and a full vector of 1s, $\vec{o}$. Recall that E^T is an M-by-N matrix, so $\vec{o}$ should be of length N, and the result will be of length M. We use a special semiring

here, `ANY_SECONDI`; the multiplicative operator will not use the value of the entries, but rather the index. Finally, we want to mask the output using the matching vector, so that unmatched edges are excluded from the result. Intuitively, the result of this operation will be an arbitrary picking of a parent vertex for each matched edge; let this be $\vec{p'}$. Next, we initialize $\vec{p}$ to be a full vector with $\vec{p}_i = i$, which handles vertices not engaged in a matching. Finally, performing $\vec{p} = E\vec{p'}$ over `ANY_SECOND` with the appropriate masking produces the desired result. Now, we can use `GrB_Matrix_build` as was done for the `A_to_E` function to build the corresponding S matrix using the parent vector. This second step is encapsulated in a `Parent_to_S` utility function. It is important to note that exactly one of the endpoint vertices of every matched edge will have degree 0 for the remainder of the coarsening scheme. As discussed in the coarsening section, we can either choose to keep this vertex in the graph or to eliminate it. From an implementation perspective, the former approach is slightly simpler since we only deal with multiplication of square matrices, while in the latter we will need to be constantly changing the size of our S matrices after determining vertices that are no longer relevant in the graph.

3. RESULTS

To test our algorithms, we developed a testing suite that allows us to evaluate algorithm performance on both quality and runtime metrics using graphs that are randomly generated according to user-provided parameters. In addition, we have run tests using graphs from the GAP benchmark suite [8], which is a well-known standard for evaluating graph processing performance. At the time of writing, we are still in the process of testing our coarsening algorithm, so the results listed here are for maximal matching.

3.1 Custom Benchmarking

We first begin by describing our custom testing suite for evaluating maximal matching. Our objective was to be able to evaluate the quality of the produced matching (i.e. how close it is to maximum), and runtime performance of the algorithm. In all cases, we first generate a random graph according to user-provided parameters. These parameters include the number of vertices in the graph, a sparse factor k which controls the average degree of vertices, and whether the graph (and matching) should be weighted. The generation of the graph is handled by a call to the utility function `LAGraph_Random_Matrix`, which builds the corresponding adjacency matrix. Additional user-provided parameters specify what type of test should be run (quality/performance) and other options that are detailed below.

3.1.1 Quality Testing

For quality testing, we provide two options: evaluation against a maximum matching, or against the results of a simple greedy (naive) algorithm. If testing on weighted matchings, the greedy algorithm attempts to add edges to the matching in sorted order depending on if a light or heavy matching is specified. When sorting, if two edges have the same weight, the one with a lower edge degree is favored (similar to the degree weighing that Luby's method uses). For unweighted matchings, the edge selection is done in arbitrary order. Evaluation against a maximum matching is more interesting as it involves exploring existing algorithms for computing exact maximum

matchings. Note that currently, we only support this option for random (unweighted) matchings. For bipartite graphs, a well-known and classic solution is maximum-flow, for example using the Ford-Fulkerson algorithm [9]. Lesser known, however, is a method discovered by Edmonds [10] that extends the augmenting path concept of Ford-Fulkerson to enable finding maximum matchings on general (non-bipartite) graphs, called the Blossom algorithm. Ford-Fulkerson runs in $O(Mf)$ for a graph with M edges and a maximum-flow of f; since $f \leq N$ if the graph has N vertices and $M \leq \frac{N(N-1)}{2}$, this gives us a $O(N^3)$ worst-case running time. Similarly, the Blossom algorithm is also $O(N^3)$. Due to this, and the fact that these algorithms are not parallelizable, for practical purposes, we can only run tests against maximum matchings on smaller graphs ($N \leq 1000$).

3.1.2 Performance Testing

The performance testing option is relatively straightforward, and compares the runtime of our algorithm against the greedy algorithm described above. The user still has the ability to choose the matching type, but we do not compare runtime performance against the maximum matching methods for obvious reasons.

Table 1: Custom quality results for $N = 1000$

Graph Type	Matching Type	k	Adversary Method	Quality
Bipartite	RM	10	Max-Flow	0.958
Bipartite	RM	10	Naive	0.985
Bipartite	RM	100	Max-Flow	0.998
Bipartite	RM	100	Naive	0.998
Bipartite	HEM	10	Naive	0.942
Bipartite	HEM	100	Naive	0.963
Bipartite	LEM	10	Naive	0.911
Bipartite	LEM	100	Naive	0.941
General	RM	10	Blossom	0.946
General	RM	10	Naive	0.989
General	RM	100	Blossom	0.986
General	RM	100	Naive	1.012
General	HEM	10	Naive	0.943
General	HEM	100	Naive	0.963
General	LEM	10	Naive	0.921
General	LEM	100	Naive	0.941

Table 1 demonstrates quality results from our custom benchmarking that we collected on random graphs with $N = 1000$. Each result was generated using the average of 10 SuiteSparse and adversary method (either naive or maximum) runs. We chose this fixed size for all tests due to the size constraints imposed by the maximum matching methods and to have a uniform comparison. The matching type refers to random matching (RM) or heavy/light matching (HEM/LEM, respectively). k is the average degree of vertices in the graph, meaning higher k is a more dense graph. The quality metric is an optimality proportion for how good the SuiteSparse result was

compared to the adversary method. For example, for weighted matchings, this would be the proportion of the matching weight SuiteSparse produced vs. the adversary method. Generally, we find very favorable results, with quality increasing for denser graphs. One peculiar and possibly questionable aspect to note is that the naive method actually produces higher quality matchings for nearly all tests, which suggests that the naive method actually produces higher quality results than the modified Luby's method SuiteSparse uses. However, as we will see, the naive method suffers a major drawback in performance since by design it is serial; as stated in 2.1.1, Luby's method allows SuiteSparse to take full advantage of parallelism. In order to see the true performance benefits that SuiteSparse offers, it helps to use graphs that are far larger than $N = 1000$, to minimize the effect of irrelevant overhead.

Table 2: Custom performance results for $N = 10^6$, bipartite, random matching

k	Threads	Performance
10	24	15.54
100	24	16.32
10	12	13.85
100	12	9.13
10	8	10.38
100	8	6.62
10	1	2.01
100	1	1.25

Table 2 showcases runs of a large bipartite graph ($N = 10^6$) at various densities and thread counts, for random matching. The performance metric here is the speedup factor that SuiteSparse was able to achieve over the naive method on the same graph. Again, each result was computed as an average across 10 SuiteSparse and naive runs. The machine used to collect this data was the

`backslash` machine at Texas A&M, which features a 12-core Intel Xeon processor, 2 threads per core. With over an order of magnitude speedup, these results illustrate the sheer power of the parallelism that SuiteSparse offers; while scratch implementations of Luby's method can be written to achieve similar performance, it would be a laborious task compared to the SuiteSparse implementation. Thus, we again reiterate the key strengths of SuiteSparse:GraphBLAS: ease of use, combined with out-of-the-box parallel performance.

3.2 GAP Benchmark

As stated above, we have also run tests on our maximal matching implementation using the GAP benchmark suite. Unlike our custom benchmarking, this benchmark does not rely on comparison against an adversary method, but rather provides insight into the absolute performance of the algorithm, which allows for easy and standardized comparison against other algorithms. We selected two graphs in the suite, `GAP-road` and `GAP-twitter` to run our tests on. The former stores the US road network; it is characterized by a large diameter and relatively uniform degree across all vertices. On the other hand, `GAP-twitter` stores the social connections on Twitter, and hence is structurally quite different from `GAP-road` - it is much more dense, low-diameter, and its degree distribution is mostly uniform except for some "celebrity" vertices with extremely high degree.

Table 3: Performance on GAP Benchmark Suite graphs

Threads	`GAP-twitter` time (s)	`GAP-road` time (s)
24	544.43	2.33
12	963.10	3.27
8	1411.79	4.23
4	2947.02	7.67
2	5461.45	13.71
1	9289.61	22.17
1 (Naive method)	7686.55	70.59

The results of the GAP runs are presented in Table 3. Again, these results were collected on

`backslash`. These results used weighted matchings (the GAP benchmark graphs are weighted). One important concern that must be addressed are the fact that these graphs are directed, and our algorithms only work for undirected graphs. Indeed, we first converted these graphs to undirected graphs using an `eWiseAdd` with A and A^T before running the tests. This may be cause for concern since this means these results are not admissible as true GAP benchmarks; however, this is inherently the case since the GAP standard defines a set of kernels (algorithms) to be tested against, which excludes maximal matching. Nevertheless, given these results and the general properties of the tested graphs, we hope that we can both offer some absolute insight into the performance of the algorithm, as well as an additional reference to evaluate improvements we make in the future.

4. CONCLUSION

4.1 Conclusion

In this work, we have discussed the mathematical background and value of linear algebraic graph algorithms. We briefly surveyed the GraphBLAS standard before exploring the topics of this work, maximal matching and coarsening of undirected graphs. We motivated the need for our algorithms via applications of graph partitioning. We then discussed the implementation of our algorithms, associated utility functions, and their performance/quality results. Overall, our results suggest that our maximal matching algorithm outperform naive implementations of maximal matching heuristics, despite lagging behind in quality by a few percentage points. In future work, we hope to develop and present results for our coarsening methods, further improve the performance of our maximal matching algorithm, as well as investigate the development of an uncoarsening algorithm in SuiteSparse to produce a complete multilevel bisection algorithm. We also hope to investigate the application of more classical, exact algorithms such as the Hungarian algorithm for maximum weight matching to allow us to generate more robust benchmarks for maximal matching.

REFERENCES

[1] J. Kepner *et al.*, "Mathematical foundations of the GraphBLAS," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–9, 2016.

[2] A. Buluç *et al.*, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 643–652, 2017.

[3] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra," *ACM Trans. Math. Softw.*, vol. 45, Dec 2019.

[4] K. Andreev and H. Räcke, "Balanced graph partitioning," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, (New York, NY, USA), p. 120–124, Association for Computing Machinery, 2004.

[5] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, (New York, NY, USA), p. 29–es, Association for Computing Machinery, 1995.

[6] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[7] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 1985.

[8] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[9] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, p. 399–404, 1956.

[10] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, p. 449–467, 1965.