LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms

Gábor Szárnyas*, David A. Bader†, Timothy A. Davis‡, James Kitchen§,
Timothy G. Mattson¶, Scott McMillan‖, Erik Welch§

*CWI Amsterdam †New Jersey Institute of Technology ‡Texas A&M University §Anaconda, Inc.
¶Parallel Computing Labs, Intel Corporation, Ocean Park, WA
‖ Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

***Abstract*—Graphs and graph algorithms can be expressed in terms of linear algebra. The GraphBLAS are a library of low level building blocks for such algorithms. The GraphBLAS target algorithm *developers*. The LAGraph project targets graph algorithm *users* with high-level algorithms common in network analysis. In this paper, we describe the first release of the LAGraph library. We describe the design decisions behind the library, the contents of the library, and performance data using the GAP benchmark suite. LAGraph, however, is much more than a library development project. It is also a project to document and analyze the full range of algorithms enabled by the GraphBLAS. To that end we have developed a compact (and hopefully intuitive) notation for describing these algorithms. In this paper, we present that notation with examples from the GAP benchmark suite.**

***Index Terms*—Graph Processing, Graph Algorithms, Graph Analytics, Linear Algebra, GraphBLAS**

I. Introduction

Graphs represent networks of relationships. They play a key role in a wide range of applications. Consequently, numerous libraries of graph algorithms exist such as igraph [9], NetworkX [3], and SNAP [17]. These libraries let programmers work with graphs without the need to master the art of crafting graph algorithms.

There are multiple ways to build libraries of graph algorithms. One approach views graphs in terms of sparse matrices and graph algorithms in terms of linear algebra. This perspective led to the GraphBLAS [18], [20]; a community effort [1] to define low level building blocks for graph algorithms as linear algebra. The GraphBLAS, however, are for graph algorithm *developers*. They are too low level for graph algorithm *users*. To focus on users and the algorithms they require, researchers from the GraphBLAS community launched the LAGraph project [19].

A major goal for the LAGraph project is to produce a library of high quality, production-worthy algorithms constructed on top the GraphBLAS library. In this paper, we describe the first release of the LAGraph library [2]. While LAGraph will eventually work with any implementation of the GraphBLAS, it is currently tied to the the SuiteSparse GraphBLAS library [10].

In this first release of LAGraph, the range of algorithms is narrow (the algorithms found in the GAP benchmark suite). With so few algorithms, we were able to focus our efforts on the key design decisions needed to establish a solid foundation we can build on for the future. Those design decisions and the rationale behind them are a key contribution of this paper. We also establish a performance baseline for our ongoing work with LAGraph with results for the GAP benchmark suite [6].

The LAGraph project is much more than a library project. Another goal for the LAGraph project is to create a repository of algorithms based on the GraphBLAS and study them in order to advance the state of the art in Graph algorithms expressed as Linear algebra. To support this goal, we created a concise notation for expressing graph algorithms in terms of the GraphBLAS. As an example of the use of this notation, we use it to describe the algorithms used in the GAP benchmark suite. We view this notation as a key contribution of this paper.

II. Design Decisions

LAGraph sits between algorithm developers, GraphBLAS implementations, and end users who want to use graph algorithms. Our overarching design principle is we want LAGraph to be easy to use, but we also need it to be flexible enough to handle advanced use cases. We don't wish to compromise performance, but when the tradeoff between convenience and performance is unavoidable, we offer both and let the user choose. LAGraph has a set of data structures, utility functions, and conventions that make it convenient for algorithm developers to write algorithms on top of GraphBLAS. By following these conventions, end users get an approachable API and consistent user experience.

We investigate the following design questions:

- data structure for representing a graph
- basic/advanced mode
- algorithm calling conventions
- error handling
- user-contributed algorithms

A. Core data structure

The primary data structure in LAGraph is the `LAGraph_Graph` which consists of primary components and cached properties. The data structure is not opaque, providing the user with full ability to access and modify all internal components, unlike the way GraphBLAS objects behave. This data structure is shown at the top of Listing 1 and defined ultimately on Line 14.

```
1  typedef struct LAGraph_Graph_struct
2  {
3      GrB_Matrix   A;      // adjacency matrix of the graph
4      LAGraph_Kind kind;   // kind of graph: directed, etc.
5
6      // cached properties
7      GrB_Matrix   AT;     // transpose of A
8      GrB_Vector   row_degree;
9      GrB_Vector   col_degree;
10     LAGraph_BooleanProperty A_pattern_is_symmetric;
11     int64_t      ndiag;  // -1 if unknown
12 } *LAGraph_Graph;
13
14 typedef struct LAGraph_Graph_struct *LAGraph_Graph;
15
16 // creating a graph
17 GrB_Matrix M;
18 // ...construction of M omitted
19
20 LAGraph_Graph G;
21 LAGraph_New(&G, &M, LAGRAPH_DIRECTED_ADJACENCY);
22
23 // operating on properties
24 LAGraph_Property_AT(G, msg);  // compute/cache
```

Listing 1: `LAGraph_Graph` data structure and methods.

The primary components of this struct are a GraphBLAS matrix named `A` and an enumeration `kind`. The kind indicates how the matrix should be interpreted. Currently, the only kinds defined are `LAGRAPH_ADJACENCY_UNDIRECTED` and `LAGRAPH_ADJACENCY_DIRECTED`, but more options will be added in the future. Creating the Graph object is performed on Line 21 of Listing 1. Following this call, `M` will be `NULL`. The matrix previously pointed to by `M` now lives at `G->A`. This "move" constructor helps avoid memory freeing errors.

Cached properties include the transpose of `A`, the row degrees, column degrees, etc. They can be computed from the primary components, but doing so repeatedly for each algorithm utilizing `A` would be wasteful. Having them live inside the Graph object helps simplify algorithm call signatures. Utility functions exist to compute each cached property. For example, Line 24 of Listing 1 will compute the transpose of `G->A` and store it as `G->AT`. Following this call, any algorithm which is given `G` will have access to both `A` and its transpose.

Because the Graph object is not opaque, any piece of code may set the transpose as well. For instance, if an algorithm computes the transpose as part of its normal logic, it could directly set `G->AT`. The expectation is that the Graph object will always remain consistent. If `G->A` is modified, all cached properties must be either be set as unknown or modified to reflect the change. Properties which are not known are set to `NULL` or `LAGRAPH_BOOLEAN_UNKNOWN` in the case of Boolean properties. This expectation is a convention that all LAGraph algorithm implementers are expected to follow.

B. User modes

Algorithms in LAGraph target two types of user modes: Basic and Advanced. The Basic user mode is for those wanting things to "just work", are less concerned about performance, and may be less experienced with the library. The Advanced user mode is for those whose primary concern is performance and are willing to conform to stricter requirements to achieve that goal.

Algorithms targeting the Basic mode typically have limited options. Often, there will only be one function for a given algorithm. Under the hood, that single algorithm might take different paths depending on the shape or size of the input graph. The idea is that a basic user wants to compute PageRank or Betweenness Centrality, but doesn't want to have to understand the five different ways to compute them. They simply want the correct answer.

Algorithms targeting the Advanced mode are often highly specialized implementations of an algorithm. The Advanced mode user is expected to understand details such as push-pull and batch mode and why different techniques are better for each graph. Advanced mode algorithms are very strict in their input. If the input doesn't match the expected kind, an error will be raised.

Advanced mode algorithms will also raise an error if a cached property is needed by an algorithm, but is not currently available on the Graph object. While Basic mode algorithms are free to compute and cache properties on the Graph object, Advanced mode algorithms never will. The idea is to never surprise the user with unexpected additional computation. An Advanced mode user must opt-in to all computations.

Often, Basic mode algorithms will inspect the input, possibly compute properties or transform the data, and finally call one of the Advanced mode algorithms to do the actual work on the graph. Having these two user modes allows LAGraph to target a wider range of users who vary in their experience with graph algorithms.

C. Algorithm calling conventions

Algorithms in LAGraph follow a general calling convention.

```
int algorithm
(
    // outputs:
    TYPE *out1,
    TYPE *out2,
    ...
    // input/output
    TYPE inout,
    ...
    // inputs
    TYPE input1,
    TYPE input2,
    ...
    // error message holder
    char *msg
)
```

The return value is always an int with the following meaning:

- `=0` -> success
- `<0` -> error
- `>0` -> warning

The meaning of a given error or warning value is algorithm-specific and should be listed in the documentation for the algorithm.

Outputs appear first and are passed by reference. A pointer should be created by the caller, but memory will be allocated by the algorithm. If the output is not needed, a `NULL` is passed and the algorithm will not return that output.

Input/Output arguments are passed by value. The expectation is that the object will be modified. This supports things like batch mode in which a frontier is updated and returned to the caller. It also supports Basic mode algorithms which may modify a Graph object by adding cached properties.

Inputs are passed by value and should never be changed by the algorithm.

The final argument of any LAGraph algorithm is the error message holder. This must be `char[]` of size `LAGRAPH_MSG_LEN`. When the algorithm returns an error or a warning, a message may be placed in this array as additional information. Because the caller creates this array, the caller must free the memory or reuse it as appropriate. If the algorithm is successful, it should fill the message array with an empty string to clear any previous message.

D. Error handling

Because every algorithm in LAGraph can return an error, the return value of every call should be checked before proceeding. To make this less burdensome for a C-based library, LAGraph provides a convenience macro which works similar to try/catch in other languages.

```
#define LAGraph_TRY(LAGraph_method)
{
    int LAGraph_status = LAGraph_method;
    if (LAGraph_status < 0)
    {
        LAGraph_CATCH (LAGraph_status);
    }
}
```

`LAGraph_CATCH` can be defined before an algorithm and will be called in the event of an error. This allows for proper freeing of memory and other necessary tasks.

A similar macro, `GrB_TRY`, will call `GrB_CATCH` when making GraphBLAS calls which return a `GrB_Info` value other than `GrB_SUCCESS` or `GrB_NO_VALUE`.

`LAGraph_TRY` and `GrB_TRY` provide an easy to use and easy to read method for dealing with error checking while writing graph algorithms.

E. Contributing algorithms

The LAGraph project welcomes contributions from all graph practitioners who understand the GraphBLAS vision of using the language of linear algebra to express graph computations. However, as a matter of practical concern, many users want a stable experience when using LAGraph for doing real work. To balance these, the LAGraph repository will have both a stable and an experimental folder.

New algorithms or modifications of existing algorithms will first be added to the experimental folder. The release schedule of experimental algorithms will generally be much faster than the stable release and there is no expectation of a bug-free experience. The goal is to generate lots of ideas and allow uninhibited contributions to push the boundary of what is possible with the GraphBLAS. The stable release will be fully tested and will move much slower, targeting the needs of those who want to use LAGraph as a complete, production-grade library rather than as a research project.

III. GraphBLAS Theory and Notation

-

In this section, we summarize the key concepts in GraphBLAS, then present a concise notation for the operations and methods defined in the GraphBLAS standard. Additionally, we demonstrate how the operations can be interpreted as graph processing primitives if graphs are encoded as adjacency matrices and nodes are selected using vectors.[1]

A. Overview

We first give a brief overview of the theoretical aspects of the GraphBLAS. For more details, we refer the reader to tutorials [22] and the specification documents [7], [12].

a) Data structures: GraphBLAS builds on the duality between the graph and matrix data structures. Namely, a directed simple graph $G = (V, E)$ can be represented with a boolean *adjacency matrix* $\mathbf{A} \in \mathbb{B}^{|V|\times|V|}$ where $\mathbf{A}_{i,j} = \texttt{TRUE}$ iff $(v_i, v_j) \in E$. The adjacency matrices used in GraphBLAS algorithms are not necessarily square: e.g., induced subgraphs, where source nodes are selected from $V_1 \subseteq V$ and target nodes are selected from $V_2 \subseteq V$, can be represented with $\mathbf{A} \in \mathbb{B}^{|V_1|\times|V_2|}$. The transposition of $\mathbf{A} \in D^{n\times m}$ is denoted with $\mathbf{A}^\mathsf{T} \in D^{m\times n}$ where $\mathbf{A}^\mathsf{T}(i, j) = \mathbf{A}(j, i)$. Compared to $\mathbf{A}$, matrix $\mathbf{A}^\mathsf{T}$ contains the edges in the reverse direction.

Vectors can be used to encode data for nodes, e.g., $\mathbf{u} \in \mathbb{B}^{|V|}$ can be used to select a subset of nodes. For vectors, $\mathbf{u}$ denotes a column vector and $\mathbf{u}^\mathsf{T}$ denotes a row vector. Vectors and matrices can be defined over different types, e.g., a (`UINT64`) matrix can encode the number of paths between two nodes, while a floating point (`FP64`) matrix can encode edge weights.

In practice, the adjacency matrices representing graphs are sparse, i.e., most of their elements are *zeros*, lending themselves to compressed representations such as CSR/CSC. The *zero elements* take their values during operations based on the identity value of the semiring's $\oplus$ operation (see below).

b) Semirings: GraphBLAS uses matrix operations to express graph processing primitives, e.g., a matrix-vector multiplication $\mathbf{A} \oplus.\otimes \mathbf{u}$ finds the incoming neighbors of the set of nodes selected by vector $\mathbf{u}$ in the graph of $\mathbf{A}$.

GraphBLAS allows users to perform the multiplication operations over an arbitrary *semiring*. In general, the multiplication operator $\otimes$ is used for combining the values of matching input elements, while the addition operator $\oplus$ defines how the results should be summarized. For example, the min.plus semiring uses plus as the multiplication operator to compute the path length and min as the addition operator to determine the length of the shortest path. The algorithms presented in this paper use

[1] We use these specialized vectors/matrices here for illustration purposes – the GraphBLAS standard allows the definition of arbitrary vectors/matrices.

operation/method	description	notation
`mxm`	matrix-matrix multiplication	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\oplus.\otimes\mathbf{B}$
`vxm`	vector-matrix multiplication	$\mathbf{w}^\mathsf{T}\langle\mathbf{m}^\mathsf{T}\rangle\odot=\mathbf{u}^\mathsf{T}\oplus.\otimes\mathbf{A}$
`mxv`	matrix-vector multiplication	$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{A}\oplus.\otimes\mathbf{u}$
`eWiseAdd`	element-wise addition using operator `op`	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\ \mathsf{op}_\cup\ \mathbf{B}$
	on elements in the set union of structures of $\mathbf{A}/\mathbf{B}$ and $\mathbf{u}/\mathbf{v}$	$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\ \mathsf{op}_\cup\ \mathbf{v}$
`eWiseMult`	element-wise multiplication using operator `op`	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\ \mathsf{op}_\cap\ \mathbf{B}$
	on elements in the set intersection of structures of $\mathbf{A}/\mathbf{B}$ and $\mathbf{u}/\mathbf{v}$	$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\ \mathsf{op}_\cap\ \mathbf{v}$
`extract`	extract submatrix from matrix $\mathbf{A}$ using indices $\boldsymbol{i}$ and indices $\boldsymbol{j}$	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}(\boldsymbol{i},\boldsymbol{j})$
	extract the jth column vector from matrix $\mathbf{A}$	$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{A}(:,j)$
	extract subvector from $\mathbf{u}$ using indices $\boldsymbol{i}$	$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}(\boldsymbol{i})$
`assign`	assign matrix to submatrix with mask for $\mathbf{C}$	$\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot=\mathbf{A}$
	assign scalar to submatrix with mask for $\mathbf{C}$	$\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot=s$
	assign vector to subvector with mask for $\mathbf{w}$	$\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot=\mathbf{u}$
	assign scalar to subvector with mask for $\mathbf{w}$	$\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot=s$
`subassign (GxB)`	assign matrix to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$	$\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot=\mathbf{A}$
	assign scalar to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$	$\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot=s$
	assign vector to subvector with submask for $\mathbf{w}(\boldsymbol{i})$	$\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot=\mathbf{u}$
	assign scalar to subvector with submask for $\mathbf{w}(\boldsymbol{i})$	$\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot=s$
`apply`	apply unary operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot=f(\mathbf{A},k)$
		$\mathbf{w}\langle\mathbf{m}\rangle\odot=f(\mathbf{u},k)$
`select`	apply select operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\langle f(\mathbf{A},k)\rangle$
		$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\langle f(\mathbf{u},k)\rangle$
`reduce`	row-wise reduce matrix to column vector	$\mathbf{w}\langle\mathbf{m}\rangle\odot=[\oplus_j\,\mathbf{A}(:,j)]$
	reduce matrix to scalar	$s\odot=[\oplus_{i,j}\,\mathbf{A}(i,j)]$
	reduce vector to scalar	$s\odot=[\oplus_i\,\mathbf{u}(i)]$
`transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}^\mathsf{T}$
`dup`	duplicate matrix	$\mathbf{C}\leftarrow\mathbf{A}$
	duplicate vector	$\mathbf{w}\leftarrow\mathbf{u}$
`build`	matrix from tuples	$\mathbf{C}\leftarrow\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\}$
	vector from tuples	$\mathbf{w}\leftarrow\{\boldsymbol{i},\boldsymbol{x}\}$
`extractTuples`	extract index arrays ($\boldsymbol{i}$,$\boldsymbol{j}$) and value arrays ($\boldsymbol{x}$)	$\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\}\leftarrow\mathbf{A}$
		$\{\boldsymbol{i},\boldsymbol{x}\}\leftarrow\mathbf{u}$
`extractElement`	extract element to scalar	$s=\mathbf{A}(i,j)$
		$s=\mathbf{u}(i)$
`setElement`	set element	$\mathbf{C}(i,j)=s$
		$\mathbf{w}(i)=s$

TABLE I: GraphBLAS operations and methods based on [11]. *Notation:* Matrices and vectors are typeset in bold, starting with uppercase ($\mathbf{A}$) and lowercase ($\mathbf{u}$) letters, respectively. Scalars including indices are lowercase italic (k, i, j) while arrays are lowercase bold italic ($\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{j}$). $\oplus$ and $\otimes$ are the addition and multiplication operators forming a semiring and default to conventional arithmetic $+$ and $\times$ operators. $\odot$ is the accumulator operator. Operations can be modified via a descriptor; matrices can be transposed ($\mathbf{B}^\mathsf{T}$), the mask can be complemented, and the mask can be valued (shown above) or structural ($\mathbf{C}\langle s(\mathbf{M})\rangle$). A structural mask can also be complemented ($\mathbf{C}\langle\neg s(\mathbf{M})\rangle$). The result can be cleared after using it as input to the mask/accumulator step ($\mathbf{C}\langle\mathbf{M},\mathrm{r}\rangle$). Not all methods are listed (creating new operators, monoids, and semirings, clearing a matrix/vector, etc.).

a number of non-conventional semirings such as any.secondi, plus.first, and plus.pair. These are summarized in Table II and defined in Sec. VI.

name	$\oplus$	$\otimes$	D	zero
conventional	plus	times	UINT64	0
any.secondi	any	secondi	UINT64	0
min.plus	min	plus	FP64	$-\infty$
plus.first	plus	first	UINT64	0
plus.second	plus	second	UINT64	0
plus.pair	plus	pair	UINT64	0

TABLE II: Semirings used in this paper

c) Masks and accumulators: All GraphBLAS operations whose output is a vector or a matrix allow the use of masks to limit the scope of the computation and an accumulator $\odot$, a binary operator that determines how the result of an operation should be applied to its output. The semantics of the masks are that the computation should be performed on a given set of nodes (for vector masks) or on a given set of edges (for matrix masks). The accumulator determines how the results should

be applied to the (potentially non-empty) output matrix/vector. The interplay of masks and the accumulators is discussed in the specifications [7], [12].

d) Notation: To present our algorithms, we use the mathematical notation given in Table I. Matrices and vectors are typeset in bold, starting with uppercase ($\mathbf{A}$) and lowercase ($\mathbf{u}$) letters, respectively. Scalars including indices are lowercase italic (k, i, j) while arrays are lowercase bold italic ($\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{j}$).

B. Operations

a) Matrix multiplication: The *matrix-matrix multiplication* operation $\mathbf{A} \oplus.\otimes \mathbf{B}$ expresses a navigation step that starts in the edges of $\mathbf{A}$ and traverses from their endpoints using the edges of $\mathbf{B}$. The result matrix $\mathbf{C}$ has elements $\mathbf{C}_{i,j}$ representing the summarized paths (e.g., number of paths, shortest paths) between start node i in the graph of $\mathbf{A}$ and end node j in the graph of $\mathbf{B}$.

The *vector-matrix multiplication* operation $\mathbf{u}^\mathsf{T} \oplus.\otimes \mathbf{A}$ performs navigation starting from the nodes selected in vector $\mathbf{u}$ among the edges of matrix $\mathbf{A}$. The result vector $\mathbf{w}$ contains the set of reached nodes with the values computed on the semiring (combining the source node values with the outgoing edge values using $\otimes$ then summarizing these for each target node using $\oplus$). The *matrix-vector multiplication* operation $\mathbf{A} \oplus.\otimes \mathbf{u}$ performs navigation in the reverse direction of the edges of $\mathbf{A}$.

b) Element-wise addition: The *element-wise addition* operations $\mathbf{u}\ \mathsf{op}_\cup\ \mathbf{v}$ and $\mathbf{A}\ \mathsf{op}_\cup\ \mathbf{B}$ applies the operator op on the elements selected by the *union of the structures of its inputs*, i.e., nodes/edges which are present in at least one of the input matrices.

c) Element-wise multiplication: The *element-wise multiplication* operation $\mathbf{u}\ \mathsf{op}_\cap\ \mathbf{v}$ and $\mathbf{A}\ \mathsf{op}_\cap\ \mathbf{B}$ applies the operator op on the elements selected by the *intersection of the structures of its inputs*, i.e., nodes/edges which are present in both inputs.

d) Extract: For adjacency matrix $\mathbf{A}$, the *extract submatrix* operation $\mathbf{A}(\boldsymbol{i},\boldsymbol{j})$ returns a matrix containing the elements from $\mathbf{A}$ with row indices in $\boldsymbol{i}$ and column indices in $\boldsymbol{j}$. In graph terms, the submatrix represents an the induced subgraph where the source nodes of the edges are in array $\boldsymbol{i}$ and the target nodes of the edges are in array $\boldsymbol{j}$. The *extract vector* operation $\mathbf{A}(i,:)$ selects a column vector containing node i's neighbors along incoming edges. The *extract subvector* operation $\mathbf{u}(\boldsymbol{i})$ selects the nodes with indices in array $\boldsymbol{i}$.

e) Assign: The *assign* operation has multiple variants. The first one assigns a matrix to a submatrix selected by row indices $\boldsymbol{i}$ and column indices $\boldsymbol{j}$: $\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j}) \odot= \mathbf{A}$. This operator is useful e.g. to "project" an induced subgraph back to the original graph. The second one assigns a vector to a selected subvector selected by indices $\boldsymbol{i}$: $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i}) \odot= \mathbf{u}$. Finally, both the selected submatrix/subvector can be assigned with a scalar value: $\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j}) \odot= s$ and $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i}) \odot= s$. In all cases, the scope of the assignment can be further constrained using masks (see Sec. III-C).

f) Apply and select: The *apply* and *select* operations evaluate a unary operator f with an optional input k (the *thunk*) on all elements of the input matrix/vector. When evaluated on a given element, function f can access the indices of the element, allowing the operation to be constrained on e.g. the lower triangle of a matrix. In the case of *apply*, denoted with $f(\mathbf{A},k)$ and $f(\mathbf{u},k)$, the resulting elements are returned as part of the output. The *select* operation requires f to be a boolean function and zeros out elements that return `FALSE`. Intuitively, $\mathbf{A}\langle f(\mathbf{A},k)\rangle$ and $\mathbf{u}\langle f(\mathbf{u},k)\rangle$ express filtering on the edges of matrix $\mathbf{A}$ and the nodes of vector $\mathbf{u}$, respectively.

g) Reduce: For adjacency matrix $\mathbf{A}$, the *row-wise reduction* $\mathbf{w}\langle\mathbf{m}\rangle \odot= [\oplus_j \mathbf{A}(:,j)]$ represents a summarization of the values on outgoing edges for each node (represented by row vector $\mathbf{A}(:,j)$) to vector $\mathbf{w}$. For matrix $\mathbf{A}$, the *reduction to scalar* $s \odot= [\oplus_{i,j} \mathbf{A}(i,j)]$ represents a summarization of all edge values. For vector $\mathbf{u}$, the *reduction to scalar* $s\odot=[\oplus_i \mathbf{u}(i)]$ represents a summarization of all node values.

h) Transposition: Transposition can be applied as a standalone GraphBLAS operation $\mathbf{C}\langle\mathbf{M}\rangle \odot= \mathbf{A}^\mathsf{T}$ and also to the input/output matrices of operations, for example:

$$\mathbf{C}^{[\mathsf{T}]}\langle\mathbf{M}\rangle \odot= \mathbf{A}^{[\mathsf{T}]} \oplus.\otimes \mathbf{B}^{[\mathsf{T}]}$$

C. Masks

Masks can be employed for all GraphBLAS operations to limit the scope of the computation w.r.t. the output of the operation. Therefore, for operations resulting in a vector, the mask is based on a vector $\mathbf{m}$ and for those resulting in a matrix, it is based on a matrix $\mathbf{M}$. Here, we only discuss matrix masks and the definitions can trivially adopted to vectors.

By default, a mask prescribes that the computation needs to be performed for elements covered by non-zero elements of the mask. However, masks can vary based on multiple aspects:

- Does the computation need to be performed on the elements selected by the mask ($\langle\mathbf{M}\rangle$) or the complement of these elements ($\langle\neg\mathbf{M}\rangle$)?
- How are existing elements of the output matrix that fall outside the selected ones treated? By default, masks use *merge* semantics, i.e., the computation can only effect elements selected by the mask, elements outside the mask are unaffected. If *replace* semantics is set, masks annihilate all elements outside the mask. This is denoted with $\langle\mathbf{M},\mathrm{r}\rangle$.
- How the elements are selected? By default, masks are *valued*, i.e., values in the mask are checked and elements with explicit zero values (e.g., 0 for plus.times) are not used considered to be part of the mask. To only consider the pattern of the mask, *structural masks* should be used, denoted with $\langle s(\mathbf{M})\rangle$.

Combining *replace semantics* and *structural masks* is possible and is denoted with $\langle s(\mathbf{M}),\mathrm{r}\rangle$

D. Methods

GraphBLAS provides methods for initializing and duplicating vectors and matrices (e.g., let: $\mathbf{w} \in \mathbb{Q}_{32}^n$ and $\mathbf{C} \leftarrow \mathbf{A}$), setting the values of individual elements ($\mathbf{w}(2) = \mathtt{TRUE}$), extracting the tuples in the form of index/value arrays from matrices/vectors and building them from tuples ($\mathbf{w} \leftarrow \{\boldsymbol{i},\boldsymbol{x}\}$ and $\{\boldsymbol{i},\boldsymbol{x}\} \leftarrow \mathbf{u}$). Additionally, methods are provided for creating

new operators, monoids, and semirings, clearing a matrix/vector, etc.

IV. Algorithms

A. Breadth-First Search (BFS)

The breadth-first search (BFS) builds on the observation that a vector-matrix multiplication $\mathbf{f}^\mathsf{T}\mathbf{A}$ expresses the navigation from the nodes selected by vector $\mathbf{f}$ in the graph represented by $\mathbf{A}$. The direction-optimizing push/pull BFS [5] is simple to express in GraphBLAS [24]. If $\mathbf{A}$ is held by row, then $\mathbf{f}^\mathsf{T}\mathbf{A}$ is a push step, while $\mathbf{Bf}$ is a pull step, where $\mathbf{B} = \mathbf{A}^\mathsf{T}$ is the explicit transpose of $\mathbf{A}$, also held by row. Other GraphBLAS libraries, e.g., GraphBLAST, store both directions and perform direction-optimization automatically [25]. The push-only BFS is shown in Alg. 1, while the push/pull BFS is Alg. 2.

The GraphBLAS BFS relies on the any.secondi semiring to compute a single step, $\mathbf{q}\langle\neg s(\mathbf{p})\rangle = \mathbf{q}^\mathsf{T}\mathbf{A}$, where $\mathbf{q}$ is the current frontier (using $\mathbf{q}$ as short for queue), $\mathbf{p}$ is the parent vector, and $\mathbf{A}$ is the adjacency matrix. This step assigns the parents of newly nodes, which do not yet have a parent, using the complemented structural mask $\langle\neg s(\mathbf{p})\rangle$.

Consider a matrix multiply for conventional linear algebra, where the $\oplus$ monoid sums a set of t entries to obtain a single scalar for computing $c_{ij} = \sum a_{ik}b_{kj}$ in the matrix multiply $\mathbf{C} = \mathbf{AB}$. The any monoid performs the reduction of t entries to a single number by merely selecting any one of the t entries as the result c_{ij}. The selection is done non-deterministically, allowing for a benign race condition. In the BFS, this corresponds to selecting any valid parent of a newly discovered node. Indeed, the creation of the any operator was inspired by Scott Beamer's `bfs.cc` method in the GAP benchmark, which has the same benign race condition. The any monoid translates the concept of this benign race condition to construct a valid BFS tree into a linear algebraic operation, suitable for implementation in GraphBLAS.

The secondi operator is the multiplicative operator in the any.secondi semiring, where the result of $a_{ik}b_{kj}$ is simply the index k in the semiring for $\mathbf{C} = \mathbf{AB}$. This gives the id of the parent node for a newly discovered node in the next frontier. The any monoid then selects any valid parent k.

Algorithm 1: Parents BFS (push-only).

Input: $\mathbf{A}$, *startVertex*

Function *ParentsBFS*
- $\mathbf{p}(startVertex) = startVertex$
- $\mathbf{q}(startVertex) = startVertex$
- **for** $level = 1$ **to** $\text{nrows}(\mathbf{A}) - 1$ **do**
  - $\mathbf{q}^\mathsf{T}\langle\neg s(\mathbf{p}^\mathsf{T}), \text{r}\rangle = \mathbf{q}^\mathsf{T}$ any.secondi $\mathbf{A}$
  - $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$
  - **if** $\text{nvals}(\mathbf{q}) = 0$ **then**
    - return

Algorithm 2: Direction-Optimizing Parent BFS.

Input: $\mathbf{A}, \mathbf{A}^\mathsf{T}$, *startVertex*

Function *DirectionOptimizingBFS*
- $\mathbf{q}(startVertex) = 0$
- **for** $level = 1$ **to** $\text{nrows}(\mathbf{A}) - 1$ **do**
  - **if** $Push(\mathbf{A}, \mathbf{q})$ **then**
    - $\mathbf{q}^\mathsf{T}\langle\neg s(\mathbf{p}^\mathsf{T}), \text{r}\rangle = \mathbf{q}^\mathsf{T}$ any.secondi $\mathbf{A}$
  - **else**
    - $\mathbf{q}\langle\neg s(\mathbf{p}), \text{r}\rangle = \mathbf{A}^\mathsf{T}$ any.secondi $\mathbf{q}$
  - $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$
  - **if** $\text{nvals}(\mathbf{q}) = 0$ **then**
    - return

Algorithm 3: Betweenness centrality.

Function *BrandesBC*
- // $\mathbf{P}(k, j)$ = # paths from kth source to node j
- // $\mathbf{F}$: # paths in the current frontier
- let: $\mathbf{P} \in \mathbb{Q}_{64}^{ns \times n}$
- let: $\mathbf{F} \in \mathbb{Q}_{64}^{ns \times n}$
- $\mathbf{P}(1:k, \boldsymbol{s}) = 1$
- // First frontier:
- $\mathbf{F}\langle\neg s(\mathbf{P})\rangle = \mathbf{P}$ plus.first $\mathbf{A}$
- // BFS phase:
- **for** $d = 0$ **to** $\text{nrows}(\mathbf{A})$ **do**
  - let: $\mathbf{S}[d] \in \mathbb{B}^{ns \times n}$
  - $\mathbf{S}[d]\langle s(\mathbf{F})\rangle = 1$ // $\mathbf{S}[d]$ = pattern of $\mathbf{F}$
  - $\mathbf{P}$ += $\mathbf{F}$
  - $\mathbf{F}\langle\neg s(\mathbf{P}), \text{r}\rangle = \mathbf{F}$ plus.first $\mathbf{A}$
  - **if** $\text{nvals}(\mathbf{F}) = 0$ **then**
    - break
- // Backtrack phase:
- let: $\mathbf{B} \in \mathbb{Q}_{64}^{ns \times n}$
- $\mathbf{B}(:) = 1.0$
- let: $\mathbf{W} \in \mathbb{Q}_{64}^{ns \times n}$
- **for** $i = d - 1$ **downto** 0 **do**
  - $\mathbf{W}\langle s(\mathbf{S}[i]), \text{r}\rangle = \mathbf{B}$ div$_\cap$ $\mathbf{P}$
  - $\mathbf{W}\langle s(\mathbf{S}[i-1]), \text{r}\rangle = \mathbf{W}$ plus.first $\mathbf{A}^\mathsf{T}$
  - $\mathbf{B}$ += $\mathbf{W} \times_\cap \mathbf{P}$
- // $\mathbf{centrality}(j) = \sum_i(\mathbf{B}(i, j) - 1)$
- $\mathbf{centrality}(:) = -ns$
- $\mathbf{centrality}$ += $[+_i \mathbf{B}(i, :)]$

B. Betweenness Centrality (BC)

The vertex betweenness-centrality metric is based on the number of shortest paths through any given node, $\sum_{s \neq i \neq t} \sigma(s, t|i)/\sigma(s, t)$, where $\sigma(s, t)$ is the total number of shortest paths from node s to t, and $\sigma(s, t|i)$ is the total number of shortest paths from node s to t that pass through node i. This is expensive to compute, so in practice, a subset of source nodes are chosen at random (a *batch*), of size ns.

Like the BFS, direction-optimization is incredibly simple

to add to the LAGraph algorithm for batched betweenness-centrality (BC). It only requires a simple heuristic to determine which direction to use, followed by masked matrix-matrix multiplication with the matrix or its transpose: $\mathbf{F}\langle\neg s(\mathbf{P})\rangle = \mathbf{F}\mathbf{B}^{\mathsf{T}}$ (the pull) or $\mathbf{F}\langle\neg s(\mathbf{P})\rangle = \mathbf{F}\mathbf{A}$ (the push), where $\mathbf{A}$ is the adjacency matrix of the graph and $\mathbf{B} = \mathbf{A}^{\mathsf{T}}$ is its explicit transpose, $\mathbf{F}$ is the frontier, and the complemented structural mask $\neg s(\mathbf{P})$ is the set of unvisited nodes. The multiplication $\mathbf{F}\mathbf{B}^{\mathsf{T}}$ relies on the descriptor to represent the transpose of $\mathbf{B}$, which is not explicitly transposed. In the backward phase, the pull step is $\mathbf{W} = \mathbf{W}\mathbf{A}^{\mathsf{T}}$ while the push is $\mathbf{W} = \mathbf{W}\mathbf{B}$, where $\mathbf{W}$ is the ns-by-n matrix in which centrality is accumulated (where $ns = 4$ is a typical batch size).

To simplify the presentation of the entire BC algorithm, Alg. 3 does not show the direction-optimization. It is the same transformation as the pair of BFS algorithms, where the push-only step (line 5 of Alg. 1), is expanded to a push/pull hueristic (lines 4-7 of Alg. 2).

C. PageRank (PR)

PageRank (PR) computes the importance of each node as a recursively-defined metric: a web page is important if important pages link to it. Alg. 4 shows the GraphBLAS implemenation of PR as specified in the GAP Benchmark. It uses the plus.second semiring, where $\mathsf{second}(x, y) = y$, so it can ignore any edge weights in the input matrix. The PR in GAP does not properly handle dangling vertices in the graph. The Graphalytics benchmark has a PageRank variant which avoids this problem [14]. We have included this version to compare its performance with the GAP benchmark algorithm `pr.cc`.

Algorithm 4: PageRank (as specified in the GAPBS).

Data: $\mathbf{A} \in \mathbb{B}^{n\times n}$ // adjacency matrix
damping // damping factor
tol // stopping tolerance
itermax // maximum number of iterations
Result: $\mathbf{r} \in \mathbb{Q}^n$
1 **Function** *PageRank*
2 $teleport = \frac{1-\alpha}{n}$
3 $\mathbf{r}(0:n-1) = \frac{1}{n}$, $\mathbf{t} = \mathbb{Q}^n$
4 $\mathbf{d_{out}} = [+_j\, \mathbf{A}(:,j)]$ // precomputed rowdegree
5 $\mathbf{d} = \mathbf{d_{out}}\ \mathsf{div}_{\cap}\ damping$ // prescale with damping
6 **for** $k = 1$ **to** *numIterations* **do**
7 swap $\mathbf{t}$ and $\mathbf{r}$ // t is now the prior rank
8 $\mathbf{w} = \mathbf{t}\ \mathsf{div}_{\cap}\ \mathbf{d}$
9 $\mathbf{r}(0:n-1) = teleport$
10 $\mathbf{r}$ += $\mathbf{A}^{\mathsf{T}}$ plus.second $\mathbf{w}$
11 $\mathbf{t} -= \mathbf{r}$
12 $\mathbf{t} = abs(\mathbf{t})$
13 **if** $[+_i\, \mathbf{t}(i)] < tol$ **then**
14 return // since 1-norm of change is small

D. Single-Source Shortest-Paths (SSSP)

A Delta-Stepping Single-Source-Shortest-Path algorithm in GraphBLAS is shown in Alg. 5. It relies on the min.plus semiring. Since it is a fairly complex algorithm, refer to [21] for a description of the method.

Algorithm 5: SSSP (delta-stepping).

Data:
$\mathbf{A}, \mathbf{A_H}, \mathbf{A_L} \in \mathbb{Q}^{|V|\times|V|}$
$s, i \in$ UINT64
$\Delta \in \mathbb{Q}$
$\mathbf{t}, \mathbf{t_{Req}} \in \mathbb{Q}^{|V|}$
$\mathbf{t_{B_i}}, \mathbf{e} \in$ UINT64$^{|V|}$
1 **Function** *DeltaStepping*
2 $\mathbf{A_L} = \mathbf{A}\langle 0 < \mathbf{A} \le \Delta\rangle$
3 $\mathbf{A_H} = \mathbf{A}\langle \Delta < \mathbf{A}\rangle$
4 $\mathbf{t}(:) = \infty$
5 $\mathbf{t}(s) = 0$
6 **while** $\mathrm{nvals}(\mathbf{t}\langle i\Delta \le \mathbf{t}\rangle) \neq 0$ **do**
7 $s = 0$
8 $\mathbf{t_{B_i}} = \mathbf{t}\langle i\Delta \le \mathbf{t} < (i+1)\Delta\rangle$
9 **while** $\mathbf{t_{B_i}} \neq 0$ **do**
10 $\mathbf{t_{Req}} = \mathbf{t} \times_{\cap} \mathbf{t_{B_i}}$
11 $\mathbf{t_{Req}} = \mathbf{A_L}^{\mathsf{T}}$ min.plus $\mathbf{t_{Req}}$
12 $\mathbf{e} = \mathbf{t}\langle 0 < \mathbf{e} \oplus \mathbf{t_{B_i}}\rangle$
13 $\mathbf{t_{B_i}} = \mathbf{t}\langle i\Delta \le \mathbf{t_{Req}} < (i+1)\Delta\rangle$
14 $\mathbf{t_{B_i}} = \mathbf{t_{B_i}}\langle \mathbf{t_{Req}} < t\rangle$
15 $\mathbf{t} = \mathbf{t}\ \mathsf{min}_{\cup}\ \mathbf{t_{Req}}$
16 $\mathbf{t_{Req}} = \mathbf{A_H}^{\mathsf{T}}$ min.plus $(\mathbf{t} \times_{\cap} \mathbf{e})$
17 $\mathbf{t} = \mathbf{t}\ \mathsf{min}_{\cup}\ \mathbf{t_{Req}}$
18 $i = i + 1$

E. Triangle Counting (TC)

The triangle counting (TC) problem is to compute the number of unique cliques of size 3 in a graph. The TC algorithm is shown in Alg. 6, based on [23]. It starts with a heuristic that decides when to sort the input graph by ascending

Algorithm 6: Triangle counting.

Data: $\mathbf{A} \in \mathbb{B}^{n\times n}$
Result: $t \in$ UINT64
1 **Function** *TriangleCount*
2 sample the *mean* and *median* degree of $\mathbf{A}$
3 **if** $mean > 4 \times median$ **then**
4 $\mathbf{p}$ = permutation to sort degree, ascending order
5 $\mathbf{A} = \mathbf{A}(\mathbf{p}, \mathbf{p})$
6 $\mathbf{L} = \mathrm{tril}(\mathbf{A})$
7 $\mathbf{U} = \mathrm{triu}(\mathbf{A})$
8 $\mathbf{C}\langle s(\mathbf{L})\rangle = \mathbf{L}$ plus.pair $\mathbf{U}^{\mathsf{T}}$
9 $t = [+_{ij}\, \mathbf{C}(i,j)]$

degree. Next, it constructs the lower and upper triangular part and computes a masked matrix multiply using the plus.pair semiring. Internally, a dot product method is used in SS:GrB, because $\mathbf{U}$ is transposed via the descriptor. The pair is the simple function $\mathsf{pair}(x, y) = 1$. When used in a semiring, it acts like the times operator of the conventional semiring, except that it can ignore the values of its inputs and treat them both as 1. This semiring is useful for structural computations, such as triangle counting, when the edge weights of a graph may be present but should be ignored in a particular algorithm.

F. Connected Components

The connected components algorithm in LAGraph (Alg. 7) is written by Zhang, Azad, and Buluç [26], [27]. The method maintains a forest of trees represented by a parent vector, and interatively merges trees until no more merging is possible. The method as shown in Alg. 7 is a simplified variant that it operates on the entire graph. In the LAGraph version, a subgraph is constructed first, and the method finds the connected components of the subgraph, and then operates on the entire graph.

Algorithm 7: Connected components (FastSV).

Function *FastSV*
  $n = \mathrm{nrows}(\mathbf{A})$
  $\mathbf{gf} = \mathbf{f}$
  $\mathbf{dup} = \mathbf{gf}$
  $\mathbf{mngf} = \mathbf{gf}$
  $\{\boldsymbol{i}, \boldsymbol{x}\} \hookleftarrow \mathbf{f}$
  repeat
    // Step 1: Stochastic hooking
    $\mathbf{mngf} = \mathbf{mngf}$ min $\mathbf{A}$
    $\mathbf{mngf} = \mathbf{mngf}$ second.min $\mathbf{gf}$
    $\mathbf{f}(\boldsymbol{x}) = \mathbf{f}$ min $\mathbf{mngf}$
    // Step 2: Aggressive hooking
    $\mathbf{f} = \mathbf{f}$ min $\mathbf{mngf}$
    // Step 3: Shortcutting
    $\mathbf{f} = \mathbf{f}$ min $\mathbf{gf}$
    // Step 4: Calculate grandparents
    $\{\boldsymbol{i}, \boldsymbol{x}\} \hookleftarrow \mathbf{f}$
    $\mathbf{gf} = \mathbf{f}(\boldsymbol{x})$
    // Step 5: Check termination
    $\mathbf{diff} = \mathbf{dup} \neq \mathbf{gf}$
    $sum = [+_i \mathbf{diff}(i)]$
    $\mathbf{dup} = \mathbf{gf}$
  until $sum == 0$

V. Utility Fuctions

LAGraph includes a set of utility functions that operate on a graph. All function names are prefixed with `LAGraph_` so we exclude that prefix in the names below, for brevity.

- **Graph Properties:** An `LAGraph_Graph` includes cached properties which can be assigned by Basic methods, or which are required by Advanced methods. `DeleteProperties` clear all properies, `Property_AT` computes the transpose of the adjacency matrix `G->A`, `Property_RowDegree` computes the row degrees of `G->A`, `Property_ColDegree` computes the column degrees of `G->A`, and `Property_ASymmetricPattern` determines if the pattern of `G->A` is symmetric or unsymmetric.
- **Display and debug:** `CheckGraph` checks the validity of a graph. Since the graph is not opaque, a user application is able to change a graph arbitrarily and thus might make it an invalid object. `DisplayGraph` displays a graph and its properties.
- **Memory management:** Wrappers for `malloc`, `calloc`, `realloc`, and `free`, allowing a user application to select the memory manager to be used. These default to the ANSI C11 library functions.
- **Graph I/O:** `BinRead` and `BinWrite` read/write a `GrB_Matrix` in binary form. `MMRead` and `MMWrite` read-/write a `GrB_Matrix` in Matrix Market form.
- **Matrix operations:** `Pattern` returns a boolean matrix containing the pattern of a matrix. `IsEqual` determines if two matrices are equal. It selects the appropriate `GrB_EQ_T` operator that matches the matrix type, and then calls `IsAll`. `IsAll` compares two matrices and returns false if the pattern of the two matrices differ. It then uses a given comparator operator to compare all pairs of entries, and returns true if all comparisons return true.
- **Degree operations:** `SortByDegree` returns a permutation that sorts a graph by its row/column degrees, and `SampleDegree` computes a quick estimate of the mean and median row/column degrees.
- **Error handling:** `LAGraph_TRY` and `GrB_TRY` are helper macros for a simple try/catch mechanism. They require the user application to define `LAGraph_CATCH` and `GrB_CATCH`.
- **Other:** `TypeName` returns a string with the name of a `GrB_Type`. `KindName` returns a string with of graph kind (directed or undirected). `Tic` and `Toc` provide a portable timer. `Sort1`, `Sort2`, and `Sort3` sort 1, 2, or 3 integer arrays.

VI. Evaluation

The performance of LAGraph can only be considered in context of an implementation of the underlying GraphBLAS library. This is discussed in Section VI-A, followed by performance results of the new LAGraph API on the 6 algorithms in the GAP Benchmark [5].

A. SuiteSparse Extensions

In a prior paper ([4]), an early draft of SuiteSparse:GraphBLAS v4.0.0 (Aug 2020) was compared with the GAP benchmark [5] and four other graph libraries. This prior version of SS:GrB included two primary data structures for its sparse matrices: compressed sparse vector, and a hypersparse variant [8], both held by row or by column. It included a draft implementation of a bitmap data structure that could

only be used in a prototype breadth-first search. Since then, SuiteSparse:GraphBLAS v4.0.3 has been released, with full support for bitmap and full matrices for all its operations. In an m-by-n bitmap matrix, the values are held in a full array of size mn, and another `int8_t` array of size mn holds the sparsity pattern of the matrix. A full matrix is a simple dense array of size mn.

The bitmap format is particularly important for the "pull" phase of an algorithm, as used in direction-optimizing breadth-first-search [5], [24]. The GAP benchmark suite uses this method by holding its frontier as a bitmap in the pull step and as a list in the push step. The GAP BFS was shown to be typically the fastest BFS amongst the 6 graph libraries compared in [4] (for 4 of the 5 benchmark graphs). With the addition of the bitmap format to SS:GrB, LAGraph+SS:GrB is able to come within a factor of 2 or so of the performance of the highly-tuned BFS GAP benchmark (see the results in the next section), for those 4 graphs. At the same time, however, the BFS is very easily expressed in LAGraph as an easy-to-read and easy-to-write code. This enables non-experts to obtain a reasonably high level of performance with modest programming effort when writing their own graph algorithms.

Additional optimizations added to SS:GrB in the past year include a *lazy sort*. Normally, SS:GrB keeps its vectors sorted (row vectors in a CSR matrix, or column vectors if the matrix is held by column), with entries sorted in ascending order of column or row index, respectively. This simplifies the many algorithms that operate on a `GrB_Matrix`. However, some algorithms naturally produce a jumbled result (matrix multiply in particular), while many algorithms are tolerant of jumbled input matrices. We thus allow the sort to be left pending. The lazy sort joins two other kinds of pending work in SS:GrB: *pending tuples* and *zombies* [11]. A pending tuple is an entry that is held inside a matrix in an unsorted list, awaiting insertion into the CSR/CSC format of a `GrB_Matrix`. A zombie is the opposite: it is an entry in the CSR/CSC format that has been marked for deletion, but has not yet been deleted from the matrix. With the lazy sort, the sort is postponed until another algorithm requires sorted input matrices. If the sort is lazy enough, it might never occur, which is the case for the LAGraph BFS and BC.

Positional binary operators have also been added, such as the any.secondi semiring, which makes the BFS much faster.

B. Performance Results

Our benchmark environment is an NVIDIA DGX Station (donated to Texas A&M by NVIDIA in support of this research). It includes a 20-core Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, with 40 threads. All codes were compiled with gcc 5.4.0 (-O3). All default settings were used, which means that hyperthreading was enabled. The system has 256GB of RAM in a single socket (no NUMA effects). LAGraph (Feb 15, 2021) and SuiteSparse:GraphBLAS 4.0.4-draft (Feb 15, 2021) were used. The NVIDIA DGX Station includes four P100 GPUs, but no GPUs were used by this experiment (a GPU-accelerated SS:GrB is in progress). Table III lists the run time (in seconds) for the GAP benchmark and LAGraph+SS:GrB for the 6 algorithms on the 5 benchmark matrices. The benchmark matrices are listed in Table IV.

Algorithm : package	graph, with run time in seconds				
	Kron	Urand	Twitter	Web	Road
BC : GAP	31.52	46.36	10.82	3.01	1.50
BC : SS	23.61	32.69	9.25	8.20	34.40
BFS : GAP	.31	.58	.22	.34	.25
BFS : SS	.52	1.22	.33	.66	3.32
PR : GAP	19.81	25.29	15.16	5.13	1.01
PR : SS	22.17	27.71	17.21	9.30	1.34
CC : GAP	.53	1.66	.23	.22	.05
CC : SS	3.36	4.47	1.47	1.97	.98
SSSP : GAP	4.91	7.23	2.02	.81	.21
SSSP : SS	17.37	25.54	8.54	9.61	46.79
TC : GAP	374.08	21.83	79.58	22.18	.03
TC : SS	917.99	34.01	239.58	34.65	.23

TABLE III: Run time of GAP and LAGraph+SS:GrB

graph	nodes	entries in A	graph kind
Kron	134,217,726	4,223,264,644	undirected
Urand	134,217,728	4,294,966,740	undirected
Twitter	61,578,415	1,468,364,884	directed
Web	50,636,151	1,930,292,948	directed
Road	23,947,347	57,708,624	directed

TABLE IV: Benchmark matrices (https://sparse.tamu.edu/GAP)

With the simple addition of the bitmap (needed for the pull step), the push/pull optimization in BC resulted in a nearly 2x performance gain in the GraphBLAS method for the largest matrices, as compared to the SS:GrB version used for the results presented in [4].

With this change the BC method in LAGraph+SS:GrB is not only expressible in a simple, elegant code, but it is also faster than the highly-tuned GAP benchmark method, `bc.cc`, for the three largest matrices (1.3x for Kron, 1.5x for Urand, and 1.2x for Twitter).

With the addition of bitmap format (which makes push/pull optimization very simple to express, and very fast) and the any.secondi semiring, the BFS of a directed or undirected graph is easily expressed in GraphBLAS, and has a performance that is only about 1.5x to 2x slower than the GAP benchmark. We expect the remaining performance gap arises from two issues:

1) The GAP assumes that the graph has fewer than 2^{32} nodes and edges, and thus uses 32-bit integers throughout. GraphBLAS is written for larger problems, and thus relies solely on 64-bit integers. This cannot be easily changed in GraphBLAS, but rather than "fixing" GraphBLAS to use smaller integers, it is the GAP algorithms that would need to be updated for larger graphs in the future. In the current GAP benchmark graphs, two graphs are chosen with almost exactly 4 billion edges. Graphs of current interest in large data science can easily exceeed 2^{32} nodes and edges [15].
2) In GraphBLAS, the BFS must be expressed as two calls. The first computes $\mathbf{q}\langle\neg\mathbf{p}\rangle = \mathbf{q}^{\mathsf{T}}\mathbf{A}$, and the second updates the parent vector, $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$:

```
GrB_vxm (q, p, NULL, semiring, q, A, GrB_DESC_RSC) ;
```

```
GrB_assign (p, q, NULL, q, GrB_ALL, n, GrB_DESC_S) ;
```

In Scott Beamer's `bfs.cc`, these two steps are fused, and the matrix-vector multiplication can write its result directly into the parent vector `p`. This could be implemented in a future GraphBLAS library, since the GraphBLAS API allows for a non-blocking mode where work is queued and done later, thus enabling a fusion of these two steps. SS:GrB exploits the non-blocking mode (for its lazy sort, pending tuples, and zombies) but does not *yet* exploit the fusion of `GrB_vxm` and `GrB_assign`. We intend to exploit this in the future.

Note that for the Road graph, LAGraph+SS:GrB is quite slow for all but PageRank (PR). The primary reason for this is the high diameter of the Road graph (about 6980). This requires 6980 iterations of GraphBLAS in the BFS, each with a tiny amount of work. Each call to GraphBLAS does several `malloc` and `free`s, and in some cases the workspace must be initialized. A future version of SS:GrB is planned that will eliminate this work entirely, by implementing an internal memory pool. There may be other overheads, but we hope that a memory pool, fusion to fully exploit non-blocking mode, and other optimizations will address this large performance gap for the Road graph for these algorithms.

LAGraph+SS:GrB is also up to 3x slower than the GAP for the triangle counting problem (for all but the Road graph, where it is even slower). This performance gap can be eliminated entirely in the future, if the `GrB_mxm` and `GrB_reduce` are combined in a single fused step, by a full exploitation of the GraphBLAS non-blocking mode. The current method computes $\mathbf{C}\langle s(\mathbf{L})\rangle = \mathbf{L}\mathbf{U}^{\mathsf{T}}$, followed by the reduction of $\mathbf{C}$ to a single scalar. The matrix $\mathbf{C}$ is then discarded. All that GraphBLAS needs is a fused kernel that does not explicitly instantiate the temporary matrix $\mathbf{C}$. This is permitted by the GraphBLAS C API Specification, but not yet implemented in SS:GrB.

VII. Conclusion

In this paper we have introduced the LAGraph library, the rationale behind our design decisions and have established a performance baseline based on the GAP benchmark suite. This will allow us to track performance enhancements as the library evolves. We also introduced a notation for describing graph algorithms expressed in terms of linear algebra. It is our hope that this notation will lead to a lively discussion leading to a consensus notation the larger "Graphs as Linear Algebra" community might adopt.

This is very much a foundational paper to support our ongoing work one the LAGraph project. We plan to explore Python wrappers for LAGraph that work well with workflows common in the data analytics community. In addition to the GAP benchmark suite, which is focussed on a key set of graph algorithms, we will next investigate end-to-end workflows based on the LDBC Graphalytics benchmark [13].

Algorithmically we see a number of research directions to pursue. With end-to-end workflows, the performance of data ingestion heavily impacts overall performance. We are interested in improving data ingestion performance by exploiting the CPUs SIMD instructions [16]. We are also interested in how the LAGraph algorithms map onto the GPU using future versions of the GraphBLAS optimized for GPUs.

Acknowledgements

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM21-0xxx]. G. Szárnyas was partially supported by the SQIREL-GRAPHS NWO project. T. Davis was supported by NSF CNS-1514406, NVIDIA, Intel, MIT Lincoln Lab, Redis Labs, and IBM.

References

[1] "GraphBLAS forum," https://graphblas.github.io.

[2] "LAGraph github repository," https://github.com/GraphBLAS/LAGraph.

[3] "NetworkX," in *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, R. Alhajj and J. G. Rokne, Eds. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-1-4939-7131-2_100771

[4] A. Azad *et al.*, "Evaluation of graph analytics frameworks using the GAP Benchmark Suite," in *IEEE*. IEEE, 2020, pp. 216–227. [Online]. Available: https://doi.org/10.1109/IISWC50251.2020.00029

[5] S. Beamer *et al.*, "Direction-optimizing breadth-first search," in *SC*. IEEE/ACM, 2012. [Online]. Available: https://doi.org/10.1109/SC.2012.50

[6] ——, "The GAP Benchmark Suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[7] A. Buluç *et al.*, "The GraphBLAS C API specification v1.3.0," 2019, https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf.

[8] A. Buluç and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *IPDPS*. IEEE, 2008, pp. 1–11. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536313

[9] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.org

[10] T. A. Davis, "SuiteSparse GraphBLAS repository," https://github.com/DrTimothyAldenDavis/GraphBLAS.

[11] ——, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, 2019. [Online]. Available: https://doi.org/10.1145/3322125

[12] ——. (2021) User guide for SuiteSparse:GraphBLAS, version 4.0.3. https://people.engr.tamu.edu/davis/GraphBLAS.html.

[13] A. Iosup *et al.*, "LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," *VLDB*, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1317-iosup.pdf

[14] ——, "The LDBC Graphalytics benchmark," *CoRR*, vol. abs/2011.15028, 2020. [Online]. Available: https://arxiv.org/abs/2011.15028

[15] J. Kepner *et al.*, "Multi-temporal analysis and scaling relations of 100,000,000,000 network packets," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–6.

[16] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *VLDB J.*, vol. 28, no. 6, pp. 941–960, 2019. [Online]. Available: https://doi.org/10.1007/s00778-019-00578-5

[17] J. Leskovec and R. Sosic, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1:1–1:20, 2016. [Online]. Available: https://doi.org/10.1145/2898361

[18] T. Mattson *et al.*, "Standards for graph algorithm primitives," in *HPEC*. IEEE, 2013. [Online]. Available: https://doi.org/10.1109/HPEC.2013.6670338

[19] ——, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," in *GrAPL at IPDPS*, 2019. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00053

[20] T. G. Mattson *et al.*, "GraphBLAS C API: Ideas for future versions of the specification," in *HPEC*. IEEE, 2017. [Online]. Available: https://doi.org/10.1109/HPEC.2017.8091095

[21] U. Sridhar *et al.*, "Delta-stepping SSSP: From vertices and edges to GraphBLAS implementations," in *GrAPL at IPDPS*. IEEE, 2019, pp. 241–250. [Online]. Available: https://doi.org/10.1109/IPDPSW.2019.00047

[22] G. Szárnyas, "Introduction to GraphBLAS: A linear algebraic approach for concise, portable, and high-performance graph algorithms," Dec. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4318870
[23] M. M. Wolf *et al.*, "Fast linear algebra-based triangle counting with KokkosKernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.
[24] C. Yang *et al.*, "Implementing push-pull efficiently in GraphBLAS," in *ICPP*. ACM, 2018, pp. 89:1–89:11. [Online]. Available: https://doi.org/10.1145/3225058.3225122
[25] ——, "GraphBLAST: A high-performance linear algebra-based graph framework on the GPU," *CoRR*, vol. abs/1908.01407, 2019. [Online]. Available: http://arxiv.org/abs/1908.01407
[26] Y. Zhang, A. Azad, and A. Buluç, "Parallel algorithms for finding connected components using linear algebra," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 14–27, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731520302689
[27] Y. Zhang *et al.*, "FastSV: A distributed-memory connected component algorithm with fast convergence," in *PPSC*. SIAM, 2020, pp. 46–57. [Online]. Available: https://doi.org/10.1137/1.9781611976137.5