Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra

TIMOTHY A. DAVIS, Texas A&M University

SuiteSparse:GraphBLAS is a full implementation of the GraphBLAS standard, which defines a set of sparse matrix operations on an extended algebra of semirings using an almost unlimited variety of operators and types. When applied to sparse adjacency matrices, these algebraic operations are equivalent to computations on graphs. GraphBLAS provides a powerful and expressive framework for creating graph algorithms based on the elegant mathematics of sparse matrix operations on a semiring. An overview of the GraphBLAS specification is given, followed by a description of the key features of its implementation in the SuiteSparse:GraphBLAS package.

CCS Concepts:•**Mathematics of computing** $\rightarrow$ **Graph algorithms; Mathematical software;**

Additional Key Words and Phrases: Graph algorithms, sparse matrices, GraphBLAS

ACM Reference Format:
Timothy A. Davis, 2018. Algorithm 9xx: SuiteSparse:GraphBLAS, graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.* 1, 1, Article 1 (January 2001), 15 pages.
DOI: 0000001.0000001

1. INTRODUCTION

The GraphBLAS standard defines sparse matrix and vector operations on an extended algebra of semirings. The operations are useful for creating a wide range of graph algorithms.

For example, consider the matrix-matrix multiplication, $\mathbf{C} = \mathbf{AB}$. Suppose $\mathbf{A}$ and $\mathbf{B}$ are sparse n-by-n Boolean adjacency matrices of two undirected graphs. If the matrix multiplication is redefined to use logical AND instead of scalar multiply, and if it uses the logical OR instead of add, then the matrix $\mathbf{C}$ is the sparse Boolean adjacency matrix of a graph that has an edge (i, j) if node i in $\mathbf{A}$ and node j in $\mathbf{B}$ share any neighbor in common. The OR-AND pair forms an algebraic semiring, and many graph operations like this one can be succinctly represented by matrix operations with different semirings and different numerical types. GraphBLAS provides a wide range of built-in types and operators, and allows the user application to create new types and operators. Expressing graph algorithms in the language of linear algebra provides:

— a powerful way of expressing graph algorithms with large, bulk operations on adjacency matrices with no need to iterate over individual nodes and edges,
— composable graph operations, e.g. $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$,
— simpler graph algorithms in user-code,
— simple objects for complex problems – a sparse matrix with nearly any data type,
— a well-defined graph object, closed under operations,
— and high performance: the bulk graph/matrix operations allow a serial, parallel, or GPU-based library to optimize the graph operations.

This work is supported by the National Science Foundation, under grant NSF CNS-1514406. Author address: T. Davis, Computer Science and Eng. Department, Texas A&M University.

A full and precise definition of the GraphBLAS specification is provided in *The GraphBLAS C API Specification* [Buluç et al. 2017], based on *GraphBLAS Mathematics* [Kepner 2017].

2. BASIC CONCEPTS

2.1. Graphs and sparse matrices

Many applications give rise to large graphs, with many nodes and edges. However, typical graphs are very sparse, with n nodes but only $O(n)$ edges.

Any graph $G = (V, E)$ can be considered as a sparse adjacency matrix A, either square or rectangular. The square case of an n-by-n sparse matrix is useful for representing a directed or undirected graph with n nodes, where either the matrix entry a_{ij} or a_{ji} represents the edge (i, j). In the rectangular case, an n-by-m sparse matrix can be used to represent a bipartite graph, or a hypergraph, depending on the context in which the matrix is used. Edges that do not appear in G are not represented in the data structure of the sparse matrix $\mathbf{A}$. A sparse matrix data structure allows huge graphs to be represented. A dense adjacency matrix for a graph of n nodes would take $O(n^2)$ memory, which is impossible when n is large, whereas a sparse adjacency matrix can be stored in $O(n + |E|)$ space, where $|E|$ is the number of edges present in the graph.

Values of entries not stored in the sparse data structure have some implicit value. In conventional linear algebra, this implicit value is zero, but it differs with different semirings. Explicit values are called *entries* and they appear in the data structure. The *pattern* of a matrix defines where its explicit entries appear, and can be represented as either a set of indices (i, j), or as a Boolean matrix S where $s_{ij} = 1$ if a_{ij} is an explicit entry in the sparse matrix $\mathbf{A}$.

The entries in the pattern of A can take on any value, including the implicit value, whatever it happens to be. It need not be the value zero. For example, in the max-plus tropical algebra, the implicit value is negative infinity, and zero has a different meaning.

Graph Algorithms in the Language on Linear Algebra [Kepner and Gilbert 2011], provides a framework for understanding how graph algorithms can be expressed as matrix computations. For additional background on sparse matrix algorithms, see also [Davis 2006] and a recent survey paper, [Davis et al. 2016].

3. SUITESPARSE:GRAPHBLAS METHODS AND OPERATIONS

3.1. Overview

SuiteSparse:GraphBLAS provides a collection of *methods* to create, query, and free each of its nine different types of objects: sparse matrices, sparse vectors, types, binary and unary operators, selection operators, monoids, semirings, and a descriptor object used for parameter settings. These are listed in Table I. Once these objects are created they can be used in mathematical *operations* (not to be confused with the how the term *operator* is used in GraphBLAS). The `GxB_` prefix is used for all SuiteSparse extensions to the GraphBLAS API.

The GraphBLAS API makes a distinction between *methods* and *operations*. A method is a function that works on a GraphBLAS object, creating it, destroying it, or querying its contents. An operation (not to be confused with an operator) acts on matrices and/or vectors in a semiring. Each object is described below.

3.1.1. Types. A GraphBLAS type (`GrB_Type`) can be any of 11 built-in types (Boolean, integer and unsigned integers of sizes 8, 16, 32, and 64 bits, and single and double precision floating point). In addition, user-defined scalar types can be created from nearly any C `typedef`, as long as the entire type fits in a fixed-size contiguous block of memory (of arbitrary size). All of these types can be used to create GraphBLAS sparse

Table I. SuiteSparse:GraphBLAS Objects

object	description
`GrB_Type`	a scalar data type
`GrB_UnaryOp`	a unary operator $z = f(x)$, where z and x are scalars
`GrB_BinaryOp`	a binary operator $z = f(x, y)$, where z, x, and y are scalars
`GxB_SelectOp`	a unary operator for constructing subgraphs
`GrB_Monoid`	an associative and commutative binary operator and its identity value
`GrB_Semiring`	a monoid that defines the "plus" and a binary operator that defines the "multiply" for an algebraic semiring
`GrB_Matrix`	a 2D sparse matrix of any type
`GrB_Vector`	a 1D sparse column vector of any type
`GrB_Descriptor`	parameters that modify an operation

matrices or vectors. All built-in types can typecasted as needed; user-defined types cannot.

3.1.2. Unary operators. A unary operator (`GrB_UnaryOp`) is a function $z = f(x)$. SuiteSparse:GraphBLAS comes with 67 built-in unary operators, such as $z = 1/x$ and $z = -x$, with variants for each built-in type. The user application can also create its own user-defined unary operators.

3.1.3. Binary operators. Likewise, a binary operator (`GrB_BinaryOp`) is a function $z = f(x, y)$, such as $z = x + y$ or $z = xy$. SuiteSparse:GraphBLAS provides 256 built-in binary operators, with variants for each built-in type. User-defined binary operators can also be created.

3.1.4. Select operators. The `GxB_SelectOp` operator is a SuiteSparse extension to the GraphBLAS API. It is used in the `GxB_select` operation to select a subset of entries from a matrix, like `L=tril(A)` in MATLAB. Its syntax is $z = f(i, j, m, n, a_{ij}, k)$ where a_{ij} is the numerical value of the entry in row i and column j in an m-by-n matrix. The parameter k is optional, and is used for operations analogous to the MATLAB statement `L=tril(A,k)`, where entries on or below the kth diagonal are kept and the rest discarded. The output z is true if the entry is to be kept, and false otherwise.

3.1.5. Monoids. The scalar addition of conventional matrix multiplication is replaced with a *monoid*. A monoid (`GrB_Monoid`) is an associative and commutative binary operator $z = f(x, y)$ where all three domains are the same (the types of x, y, z), and where the operator has an identity value o such that $f(x, o) = f(o, x) = x$. Performing matrix multiplication with a semiring uses a monoid in place of the "add" operator, scalar addition being just one of many possible monoids. The identity value of addition is zero, since $x+0 = 0+x = x$. GraphBLAS includes eight built-in operators suitable for use as a monoid: min (with an identity value of positive infinity), max (whose identity is negative infinity), add (identity is zero) multiply (with an identity of one), and four logical operators: AND, OR, exclusive-OR, and Boolean equality. User-created monoids can be defined with any associative and commutative operator with an identity value.

A monoid can also be used in a reduction operation, like `s=sum(A)` in MATLAB. MATLAB provides the plus, times, min, and max reductions of a real or complex sparse matrix as `s=sum(A)`, `s=prod(A)`, `s=min(A)`, and `s=max(A)`, respectively. In GraphBLAS, any monoid can be used (min, max, plus, times, AND, OR, exclusive-OR, equality, or any user-defined monoid, on any user-defined type).

3.1.6. Semirings. A *semiring* (`GrB_Semiring`) consists of an additive monoid and a multiplicative operator. Together, these operations define the matrix multiplication $\mathbf{C} = \mathbf{AB}$, where the monoid is used as the additive operator and the semiring's multiplicative operator is used in place of the conventional scalar multiplication in standard matrix multiplication via the plus-times semiring. A user application can define its own monoids and semirings.

A semiring can use any built-in or user-defined binary operator $z = f(x, y)$ as its multiplicative operator, as long as the type of its output, z matches the type of the semiring's monoid. The user application can create any semiring based on any types, monoids, and multiply operators, as long these few rules are followed.

Just considering built-in types and operators, SuiteSparse:GraphBLAS can perform `C=A*B` in 960 unique semirings. With typecasting, any of these 960 semirings can be applied to matrices `C`, `A`, and `B` of any of the 11 types, in any combination. This gives $960 \times 11^3 = 1,277,760$ possible kinds of sparse matrix multiplication supported by SuiteSparse:GraphBLAS, and this is counting just built-in types and operators. By contrast, MATLAB provides just two semirings for its sparse matrix multiplication `C=A*B`: plus-times-double and plus-times-complex, not counting the typecasting that MATLAB does when multiplying a real matrix times a complex matrix. All of the 1.3 million forms of matrix multiplication methods in SuiteSparse:GraphBLAS are typically just as fast as computing `C=A*B` in MATLAB using its own native sparse matrix multiplication methods.

3.1.7. Descriptor. A *descriptor* object, `GrB_Descriptor`, provides a set of parameter settings that modify the behavior of GraphBLAS operations, such as transposing an input matrix or complementing a `Mask` matrix (see Section 3.1.9 for details).

3.1.8. Non-blocking mode. GraphBLAS includes a *non-blocking* mode where operations can be left pending, and saved for later. This is very useful for submatrix assignment (`C(I,J)=A` where `I` and `J` are integer vectors), or or scalar assignment (`C(i,j)=x` where `i` and `j` are scalar integers). Because of how MATLAB stores its matrices, adding and deleting individual entries is very costly. For example, this is very fast in SuiteSparse:GraphBLAS, taking only $O(nz \log nz)$ time:

```
GrB_Matrix A ;
GrB_Matrix_new (&A, GrB_FP64, m, n) ;
for (int k = 0 ; k < nz ; k++)
{
    // compute a value x, row index i, and column index j
    // A (i,j) = x
    GrB_Matrix_setElement (A, x, i, j) ;
}
```

The equivalent method in MATLAB takes $O(nz^2)$ time:

```
A = sparse (m,n) ;   % an empty sparse matrix
for k = 1:nz
    compute a value x, row index i, and column index j
    A (i,j) = x ;
end
```

Building a matrix all at once via `GrB_Matrix_build` in SuiteSparse:GraphBLAS and via `sparse` in MATLAB is equally fast; both methods below take $O(nz \log nz)$ time, in SuiteSparse:GraphBLAS:

```
for (int k = 0 ; k < nz ; k++)
{
    I [k] = ... ;
    J [k] = ... ;
    X [k] = ... ;
}
GrB_Matrix A ;
```

```
GrB_Matrix_new (&A, GrB_FP64, nrows, ncols) ;
GrB_Matrix_build (A, I,J,X,nz, GrB_SECOND_FP64) ;
```

And in MATLAB:

```
I = zeros (nz,1) ;
J = zeros (nz,1) ;
X = zeros (nz,1) ;
for k = 1:nz
    compute a value x, row index i, and column index j
    I (k) = i ;
    J (k) = j ;
    X (k) = x ;
end
A = sparse (I,J,X,m,n) ;
```

Exploiting non-blocking mode, SuiteSparse:GraphBLAS can do both incremental and all-at-once methods equally fast, but in MATLAB only the all-at-once method is efficient. Allowing for fast incremental updates allows the user to write simpler code to construct a matrix, and enables much faster incremental updates to a matrix that has already been constructed.

3.1.9. The accumulator and the mask. Most GraphBLAS operations can be modified via transposing input matrices, using an accumulator operator, applying a mask or its complement, and by clearing all entries the matrix `C` after using it in the accumulator operator but before the final results are written back into it. All of these steps are optional, and are controlled by a descriptor object that holds parameter settings that control the following options:

— the input matrices `A` and/or `B` can be transposed first.
— an accumulator operator can be used, like the plus in the MATLAB statement `C=C+A*B`. The accumulator operator can be any binary operator, and an element-wise "add" (set-union) is performed using the operator.
— an optional *mask* can be used to selectively write the results to the output. The mask is a sparse Boolean matrix `Mask` whose size is the same size as the result. If `Mask(i,j)` is true, then the corresponding entry in the output can be modified by the computation. If `Mask(i,j)` is false, then the corresponding in the output is protected and cannot be modified by the computation. The `Mask` matrix acts exactly like logical matrix indexing in MATLAB, with one minor difference: in GraphBLAS notation, the mask operation is $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{Z}$, where the mask M appears only on the left-hand side. In MATLAB, it would appear on both sides as `C(Mask)=Z(Mask)`. If no mask is provided, the `Mask` matrix is implicitly all true. This is indicated by passing the value `GrB_NULL` in place of the `Mask` argument in GraphBLAS operations.

This process can be described in mathematical notation as:

$\mathbf{A} = \mathbf{A}'$, if requested via descriptor (first input option)
$\mathbf{B} = \mathbf{B}'$, if requested via descriptor (second input option)
$\mathbf{T}$ is computed according to the specific operation
$\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{C} \odot \mathbf{T}$, accumulating and writing the results back via the mask

The application of the mask and the accumulator operator is written as $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{C} \odot \mathbf{T}$ where $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ denotes the application of the accumulator operator, and $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{Z}$ denotes the mask operator via the Boolean matrix M. The expression $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{C} \odot \mathbf{T}$ is computed as follows:

```
if no accumulator operator, $\mathbf{Z} = \mathbf{T}$; otherwise $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$
if requested via descriptor (replace option), all entries cleared from $\mathbf{C}$
if Mask is NULL
    $\mathbf{C} = \mathbf{Z}$ if Mask is not complemented; otherwise $\mathbf{C}$ is not modified
else
    $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{Z}$ if Mask is not complemented;
    otherwise $\mathbf{C}\langle\neg\mathrm{M}\rangle = \mathbf{Z}$
```

The accumulator operator ($\odot$) acts like a sparse matrix addition, except that any operator can be used. The pattern of $\mathbf{C} \odot \mathbf{T}$ is the set-union of the patterns of $\mathbf{C}$ and $\mathbf{T}$, and the operator is applied only on the set-intersection of $\mathbf{C}$ and $\mathbf{T}$. Entries in neither the pattern of $\mathbf{C}$ nor $\mathbf{T}$ do not appear in the pattern of $\mathbf{Z}$. That is:

```
for all entries $(i, j)$ in $\mathbf{C} \cap \mathbf{T}$ (that is, entries in both $\mathbf{C}$ and $\mathbf{T}$)
    $z_{ij} = c_{ij} \odot t_{ij}$
for all entries $(i, j)$ in $\mathbf{C} \setminus \mathbf{T}$ (that is, entries in $\mathbf{C}$ but not $\mathbf{T}$)
    $z_{ij} = c_{ij}$
for all entries $(i, j)$ in $\mathbf{T} \setminus \mathbf{C}$ (that is, entries in $\mathbf{T}$ but not $\mathbf{C}$)
    $z_{ij} = t_{ij}$
```

3.2. GraphBLAS methods and operations

The matrix (`GrB_Matrix`) and vector (`GrB_Vector`) objects include additional methods for setting a single entry, extracting a single entry, making a copy, and constructing an entire matrix or vector from a list of *tuples*. The tuples are held as three arrays `I`, `J`, and `X`, which work the same as `A=sparse(I,J,X)` in MATLAB, except that any type matrix or vector can be constructed. A complete list of methods can be found in the User Guide provided with the software package.

Table II lists all SuiteSparse:GraphBLAS operations in the GraphBLAS notation where $\mathbf{AB}$ denotes the multiplication of two matrices over a semiring. Upper case letters denote a matrix, and lower case letters are vectors. Each operation takes an optional `GrB_Descriptor` argument that modifies the operation. The input matrices $\mathbf{A}$ and $\mathbf{B}$ can be optionally transposed, the mask M can be complemented, and $\mathbf{C}$ can be cleared of its entries after it is used in $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$ but before the $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{Z}$ assignment. Vectors are never transposed via the descriptor.

The notation $\mathbf{A} \oplus \mathbf{B}$ denotes the element-wise operator that produces a set-union pattern (like `A+B` in MATLAB). The notation $\mathbf{A} \otimes \mathbf{B}$ denotes the element-wise operator that produces a set-intersection (like `A.*B` in MATLAB). Reduction of a matrix $\mathbf{A}$ to a vector reduces the ith row of $\mathbf{A}$ to a scalar w_i, like `w=sum(A')` in MATLAB.

4. SUITESPARSE:GRAPHBLAS IMPLEMENTATION

The GraphBLAS API provides a great deal of flexibility in its implementation details. All of its objects are opaque, and their contents can only be modified by calling GraphBLAS functions. This section describes how the GraphBLAS API is implemented in the SuiteSparse:GraphBLAS software package.

4.1. Matrix and vector data structure

The GraphBLAS matrix and vector objects are opaque to the end-user, and can only be accessed via GraphBLAS methods and operations. Their implementation has a significant impact on the performance of GraphBLAS, so an overview of the data structure is given here.

In SuiteSparse:GraphBLAS (version 2.0.2), a GraphBLAS matrix (the `GrB_Matrix` object) is stored in a variant of the compressed-sparse column format. The basics of

Table II. SuiteSparse:GraphBLAS Operations

function name	description	GraphBLAS notation
`GrB_mxm`	matrix-matrix mult.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
`GrB_vxm`	vector-matrix mult.	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
`GrB_mxv`	matrix-vector mult.	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
`GrB_eWiseMult`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$
	set-union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set-intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i},\mathbf{j})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
`GrB_assign`	assign submatrix	$\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i},\mathbf{j}) = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$
		$\mathbf{w}\langle\mathbf{m}\rangle(\mathbf{i}) = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GxB_subassign`	assign submatrix	$\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$
		$\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
`GrB_apply`	apply unary op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
`GxB_select`	apply select op.	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A}, \mathbf{k})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u}, \mathbf{k})$
`GrB_reduce`	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:,j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(i,j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$
`GxB_kron`	Kronecker product	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \text{kron}(\mathbf{A}, \mathbf{B})$

this data structure are identical to the MATLAB sparse matrix [Gilbert et al. 1992], and also the sparse matrix format used in CSparse [Davis 2006]. For a matrix `A` of size `m`-by-`n`, an integer array `A.p` of size `n+1` holds the column "pointers," which are references to locations in the `A.i` and `A.x` arrays, each of which are of size equal to, or greater, than the number of entries in the matrix. The row indices of entries in column `j` of `A` are held in `A.i[A.p[j]...A.p[j+1]-1]`, and the values are held in the same positions in the `A.x` array. Row and column indices are zero-based, which matches the internal data structure of a MATLAB sparse matrix. The MATLAB user is provided the illusion of 1-based indexing in a MATLAB M-file, whereas the GraphBLAS user sees a 0-based indexing. To facilitate submatrix extraction (`C=A(I,J)`) and assignment (`C(I,J)=A`), row indices are always kept sorted, just like MATLAB. No duplicate entries are held in this format.

The type of the entries of `A` can be almost anything. GraphBLAS provides 11 built-in types (boolean, integers and unsigned integers of size 8, 16, 32, and 64 bits, and single and double precision floating-point). In addition, a user application can define any new type, without requiring a recompilation of the GraphBLAS library. The only restriction on the type is that it must have a constant size, and it must be possible to copy a value with a single `memcpy`. MATLAB provides sparse logical, double, and complex double. GraphBLAS can be easily extended to handle complex double sparse matrices via a user-defined complex type; an example is given in the `Demo` folder of SuiteSparse:GraphBLAS.

MATLAB drops its entries with a numerical value of zero, but this is never done in GraphBLAS. A "zero" is simply an entry that is not stored in the data structure, and the value of this implicit entry depends on the semiring. If the matrix is used in the conventional plus-times semiring, the implicit value is zero. If used in max-plus, the implicit entry is $-\infty$. This value is not stored in `A`; it depends on the semiring, and any matrix can be used in any semiring. Thus, dropping a numerical entry whose value happens to be zero cannot be done in GraphBLAS.

SuiteSparse:GraphBLAS adds an additional set of features to this data structure: *zombies* and *pending tuples*, which enable fast insertion and deletion of entries. These features enable submatrix assignment to be many times faster the equivalent operations in MATLAB, even when blocking mode is used.

A *zombie* is an entry in the data structure that has been marked for deletion. This is done by "negating" the row index of the entry. More precisely, its row index `i` is changed to `(-i-2)`, to accommodate zero-based row indices. Zombies allow for fast deletion, and they also permit binary searches of a column to performed, even if it contains zombies.

A *pending tuple* is an entry that has not yet been added to the compressed-sparse column part of the data structure. Pending tuples are held in an unsorted list of row indices, column indices, and values. Duplicates may appear in this list. The matrix also keeps track of a single operator to be used to combine duplicate entries. A matrix can have both zombies and pending tuples.

Temporary matrices, internal in SuiteSparse:GraphBLAS, need not have their own `p`, `i`, and/or `x` arrays. Any or all of these can be shallow pointers to the components of other matrices. This feature facilitates operations such as typecasting a user input matrix to a required type. In this case, only `x` need be constructed for the temporary typecasted matrix, while the pattern (`p` and `i`) are shallow copies of the input matrix.

A GraphBLAS vector (`GrB_Vector`) is simply held as an `n`-by-1 matrix, although the two types (`GrB_Matrix` and `GrB_Vector`) are different and unique types in the user-level application.

4.2. Matrix operations

Details of the matrix multiply, element-wise, submatrix extraction, and submatrix assignment operations are given below. Unless specified otherwise, the matrix $\mathbf{C}$ is m-by-n. The `GrB_apply`, `GrB_reduce`, `GrB_transpose`, `GxB_select`, and `GxB_kron` operations are not described here since their implementation is fairly straight-forward.

4.2.1. Matrix multiply. Multiplying two sparse matrices (`GrB_mxm`), matrix-vector multiply (`GrB_mxv`) or vector-matrix multiply (`GrB_vxm`) relies on three methods:

(1) a variant of Gustavson's algorithm [Gustavson 1978],
(2) a masked variant of Gustavson's method for $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{AB}$, and
(3) a dot-product formulation.

In the first method, when no mask is present, the work is split into a symbolic analysis phase that finds the pattern of $\mathbf{C}$ and a numerical phase that computes its values. To multiply $\mathbf{C} = \mathbf{AB}$ where $\mathbf{C}$ is m-by-n, $\mathbf{A}$ is m-by-k and $\mathbf{B}$ is k-by-n, both phases take only $O(n + f)$ time, where f is the number of "multiply-adds" computed (in the semiring). The terms m and k do not appear in the time complexity, which is very important when n and f are both much less than m or k.

A breadth-first search, for example, must compute $\mathbf{c} = \mathbf{Ab}$ where $\mathbf{b}$ is a sparse vector. The result $\mathbf{c}$ is the set of neighbors of all nodes i where b_i is true. This set of neighbors is much smaller than the number of nodes, m, in the graph, but the time to compute this set is simply $O(f)$, or the sum total of the adjacency sets for all nodes in the current level, represented by $\mathbf{b}$ (a column vector, so $n = 1$).

To obtain this time $O(n + f)$ that does not depend on m, SuiteSparse:GraphBLAS maintains a set of pre-allocated workspaces of size $O(m)$, used internally and reused for subsequent operations. One is uninitialized, and use for numerical gather/scatter operations in the numerical phase. The other is an initialized integer array, `mark`, that is used for the set-union computation in the symbolic phase, and it is cleared in constant time (when initialized, `mark[i]<flag` holds for all `i`; to set `mark[i]` as true, `mark[i]=flag` is done, and clearing the entire `mark` array simply requires `flag` to be incremented). Both workspaces are resized if they are not large enough, but this would be done just once for the entire breadth-first search, for example. Thus, in an amortized

sense, these size m arrays do not contribute an $O(m)$ component to the time complexity of computing $\mathbf{C} = \mathbf{AB}$.

Both $\mathbf{C}$ and $\mathbf{B}$ have n columns, and even if many of those columns are entirely empty, the time complexity of $O(n+f)$ still depends on n. Constructing $\mathbf{C}$ takes $\Omega(n)$ time and space since it is stored in compressed sparse-column form with a pointer array `C.p` of size $n + 1$. The term n could dominate in some cases, if n is much larger than f. In a future version of SuiteSparse:GraphBLAS, a hypersparse version of the data structure is being considered, where only non-empty columns would be counted in the memory space and time complexity of $\mathbf{C} = \mathbf{AB}$ [Buluç and Gilbert 2008]. This change can easily be done in the future since all GraphBLAS objects are opaque to the user application.

The result of computing the matrix product $\mathbf{AB}$ may be written into $\mathbf{C}$ via a mask, M, via $\mathbf{C}\langle\mathrm{M}\rangle = \mathbf{AB}$. If the mask is present (and not complemented), only the subset of entries appearing in the mask are computed. This greatly reduces the time and memory usage. In this method, the symbolic analysis is skipped. A matrix $\mathbf{T} = \mathbf{AB}$ is computed whose pattern is assumed to be a subset of the mask matrix M. Entries in $\mathbf{AB}$ outside the mask need not be computed, and are discarded if they are computed.

For example, if $\mathbf{L}$ is the strictly lower triangular part of an unweighted graph $\mathbf{A}$, then $\mathbf{C}\langle\mathbf{L}\rangle = \mathbf{L}^2$ finds the number of triangles in the graph, where c_{ij} is the number of triangles the containing the edge (i, j) [Azad et al. 2015]. Not all of $\mathbf{L}^2$ is computed or stored, but only the entries corresponding to entries in the mask, $\mathbf{L}$. This greatly reduces the time and memory complexity of the masked matrix multiply, as compared with computing all of $\mathbf{L}^2$ first and then applying the mask.

Currently, matrix multiply and submatrix assignment (`GrB_mxm`, `GrB_mxv`, `GrB_vxm`, `GrB_assign`, and `GxB_subassign`) are the only operations that exploit the mask during computations. All other operations compute their result and only then do they write their results into the output matrix via the mask.

4.2.2. Element-wise operations. Element-wise operations in GraphBLAS can use any binary operator. They are applied in either a set-union (`GrB_eWiseAdd`) or set-intersection (`GrB_eWiseMult`) manner. The set-union is like a sparse matrix addition, `C=A+B`. If an entry appears in just `A` or `B`, the value is copied into `C` without applying the binary operator. The set-intersection is like `C=A.*B` in MATLAB, which is the Hadamard matrix product if the binary multiply operator is used. For both methods, however, any binary operator can be used, which is only applied to entries in the set-intersection.

The implementation of `GrB_eWiseAdd` is straightforward, taking $O(n+|\mathbf{A}|+|\mathbf{B}|)$ time where $|...|$ denotes the number of entries in a matrix and where n is the number of columns in the three matrices. Each column of $\mathbf{C}$ is computed with a merge of the corresponding columns of $\mathbf{A}$ and $\mathbf{B}$, since columns are always kept with sorted row indices. Unless the matrices have fewer than $O(n)$ entries, this time is optimal.

The `GrB_eWiseMult` operation is more subtle, since the lower bound on the ideal time is the size of the set-intersection, $\Omega(|\mathbf{C}|)$, which is smaller than either $|\mathbf{A}|$ or $|\mathbf{B}|$. For each column j, if the number of nonzeros in $\mathbf{A}_{*j}$ and $\mathbf{B}_{*j}$ are similar, then a conventional merge is used, taking $O(|\mathbf{A}_{*j}| + |\mathbf{B}_{*j}|)$ time. This is just like set-union "add," except that entries outside the set-intersection are not copied into $\mathbf{C}$. Suppose instead that the jth column of $\mathbf{A}$ has far fewer entries than the jth column of $\mathbf{B}$. In this case, during the merge, each entry of $\mathbf{A}_{ij}$ is examined and a trimmed binary search is used to find the corresponding entry in $\mathbf{B}_{*j}$. The search is trimmed since $\mathbf{B}_{*j}$ is sorted, and once an entry i is found this result is used to reduce the search for subsequent row indices $i' > i$.

Additional tests for special cases reduce the time even further. For example, if the last entry in column $\mathbf{A}_{*j}$ comes before the first entry in $\mathbf{B}_{*j}$, then the intersection is empty and no traversal is done of either $\mathbf{A}_{*j}$ or $\mathbf{B}_{*j}$.

These tests are performed column-by-column. In the extreme case, if $\mathbf{A}$ is very sparse and $\mathbf{B}$ is completely dense, the time complexity of `GrB_eWiseMult` is $O(n + |\mathbf{A}| \log m)$. This is much smaller than the $O(n^2)$ time that would result if a simple merge is used, like the set-union "add" method used in `GrB_eWiseAdd`.

4.2.3. Submatrix extraction. `GrB_extract` extracts a submatrix, $\mathbf{C} = \mathbf{A}(\mathbf{i}, \mathbf{j})$ where $\mathbf{i}$ and $\mathbf{j}$ are integer vectors, like `C=A(I,J)` in MATLAB notation. It is a meta-algorithm that selects different methods for traversing the matrix $\mathbf{A}$, depending on the length $|\mathbf{i}|$ of the row index list $\mathbf{i}$ and the number of entries in a given column of the matrix $\mathbf{A}$.

The matrix `C` is built one column `C(:,k)` at a time, by examining `A(:,j)` where `j=J[k]`. Let $a = |\mathbf{A}_{*j}|$ be the number of entries in the jth column of $\mathbf{A}$, and let $c = |\mathbf{C}_{*k}|$ be the number of entries in the kth column of $\mathbf{C}$. If `I` has duplicate entries, then $c > a$ is possible; otherwise $c \leq a$ must hold.

If the list `I` is long, and not contiguous (nor `GrB_ALL`, which acts just like the colon in `A(:,j)` in MATLAB), then an multiset inverse of `I` is created in $O(m + |\mathbf{i}|)$ workspace, where m is the number of rows of `A`, taking $O(|\mathbf{i}|)$ time. The inverse is a multiset if `I` contains duplicate entries. This workspace is needed only for cases 5 to 7, below.

Except for case 2, the method starts with a binary search for the first row index, `I[0]`, taking at most $O(\log a)$ time. The method then considers seven cases, for each column of `C` and `A`; the first case that holds is utilized.

(1) `I` is a single row index: the entry is extracted in $O(1)$ time. Total time is $O(\log a)$.
(2) `I` is `GrB_ALL` (the colon `C=A(:,j)` in MATLAB notation): the entire column is copied. Time is optimal: $O(c)$.
(3) `I` is a contiguous list (`C=A(imin:imax,j)`): the entire column is copied starting at `imin=I[0]` until reaching `imax`. Time is $O(\log a + c)$, which is nearly equal to the lower bound of $O(c)$.
(4) The inverse of `I` has not been computed, or the size of `I` is small compared with the number of entries in `A(:,j)`: a binary search in `A(:,j)` is performed for each entry in `I`. If found, the entry is appended to `C(:,k)`. Time is $O(|\mathbf{i}| \log a)$. Case 4 is faster than cases 5 to 7 when $|\mathbf{i}|$ is smaller than a.
(5) The list `I` is not in order (and may include duplicates): all of `A(:,j)` is examined. If `A(i,j)` is nonzero, then all row indices `t` for which `t=I[k]` (if any) are appended to the column `C(:,k)`, along with their positions in `A(:,j)`. This traversal relies on the multi-inverse of `I` computed in the initialization phase. Once the traversal of the pattern of `A(:,j)` is completed, the row indices and positions in `C(:,k)` are sorted. Next the corresponding numerical values are copied from `A(I,j)` to `C(:,k)`. Time is $(c \log c + a)$.
(6) The list `I` is sorted but has duplicates: all of `A(:,j)` is examined. This method is just like case (5), except that no sort is needed, and it is done in a single pass. Time is $O(c + a)$.
(7) the list `I` is sorted, with no duplicates: this is a simplified version of case (6). Time is $O(a)$.

There is no simple expression for the total time to construct all of $\mathbf{C} = \mathbf{A}(\mathbf{i}, \mathbf{j})$, since each column is treated differently in one of the above seven cases.

4.2.4. Submatrix assignments. The `GrB_assign` and `GxB_subassign` operations modify a submatrix of `C`. The GraphBLAS operation `GrB_assign` computes $\mathbf{C}\langle\mathbf{M}\rangle(\mathbf{i}, \mathbf{j}) = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$, where the mask matrix $\mathbf{M}$ has the same size as $\mathbf{C}$. SuiteSparse:GraphBLAS adds an extension to the GraphBLAS specification, via the operation `GxB_subassign`, which computes $\mathbf{C}(\mathbf{i}, \mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i}, \mathbf{j}) \odot \mathbf{A}$. In this case, the mask matrix is the same size as the submatrix $\mathbf{C}(\mathbf{i}, \mathbf{j})$. Details of the two operations are shown in Table III.

Table III. `GrB_assign` and `GxB_subassign`

Step	GrB_assign	GxB_subassign
1	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$	$\mathbf{S} = \mathbf{C}(\mathbf{I}, \mathbf{J})$
2	$\mathbf{S} = \mathbf{S} \odot \mathbf{A}$	$\mathbf{S}\langle\mathbf{M}\rangle = \mathbf{S} \odot \mathbf{A}$
3	$\mathbf{Z} = \mathbf{C}$	$\mathbf{C}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$
4	$\mathbf{Z}(\mathbf{I}, \mathbf{J}) = \mathbf{S}$	
5	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$	

These two operations are the most intricate operations in SuiteSparse:GraphBLAS. They fully exploit non-blocking mode to obtain high performance.

If a sequence of assignments uses the same accumulator operator, portions of the assignment are postponed, via zombies and pending tuples. In most cases, the method starts with a specialized form of submatrix extraction, $\mathbf{S} = \mathbf{C}(\mathbf{i}, \mathbf{j})$, where the scalar entries in S are pointers to where the entries reside in $\mathbf{C}$. Next $\mathbf{S} = \mathbf{S} \odot \mathbf{A}$ is computed, where entries that appear in both S and $\mathbf{A}$ are modified directly in $\mathbf{C}$. Entries that appear in S but not $\mathbf{A}$ become zombies in $\mathbf{C}$. Entries that appear in $\mathbf{A}$ but not S become pending tuples in $\mathbf{C}$.

No workspace is required beyond that needed by the submatrix extraction to compute S. As a result, if $\mathbf{C}$ is m-by-n, in most cases no size m or size n workspace is needed. This makes the method much more efficient for very large, very sparse matrices, as compared to `C(I,J)=A` in MATLAB, even when the non-blocking mode is not exploited. In particular, for one square matrix `C` of dimension 3 million, containing 14.3 million nonzeros, `C(I,J)=A` takes 87 seconds in MATLAB on a MacBook Pro but only 0.74 seconds in SuiteSparse:GraphBLAS (where `A` is 5500-by-7000 with 38,500 nonzeros). This 0.74 seconds includes the time to return the result back to MATLAB as a valid MATLAB sparse matrix with all zombies and pending tuples removed.

The `GrB_Matrix_setElement` method modifies a single entry, `C(i,j)=x` where `i` and `j` are scalars. It first performs a binary search of column `C(:,j)` to see if the entry is present. If so, the value is modified; a zombie may come back to life in this case. If it is not present, the entry is added to the list of pending tuples.

4.3. Testing

SuiteSparse:GraphBLAS includes a `Test` folder that contains a full MATLAB implementation of the GraphBLAS API Specification, so that each method and operation in the C-callable SuiteSparse:GraphBLAS can be tested and their results compared with the GraphBLAS specification.

5. EXAMPLE GRAPH ALGORITHMS IN GRAPHBLAS

This section describes two of the graph algorithms provided as demos in the SuiteSparse:GraphBLAS package: breadth-first search, and Luby's method for finding a maximal independent set.

5.1. Breadth-first search

Figure 1 gives an algorithm in GraphBLAS for computing the breadth-first search of a graph. Each step consists of a matrix-vector multiplication, which finds the nodes in the next level of the traversal. Space does not permit all the details of the C API syntax to be described, so the algorithm is repeated in pseudo-MATLAB notation in Figure 2. The single call to `GrB_mxv` computes a masked matrix-vector product using the logical OR-AND semiring. Internally, it first computes `t=A*q` using the Boolean OR-AND semiring. It then clears all entries in `q`, via the `GrB_REPLACE` parameter in the descriptor. Finally, it assigns `t` to `q` using a complemented mask, which is `q(~v)=t` in MATLAB notation.

```
GrB_Info bfs                // BFS of a graph (using vector assign & reduce)
(
    GrB_Vector *v_output,   // v [i] is the BFS level of node i in the graph
    const GrB_Matrix A,     // input graph, treated as if boolean in semiring
    GrB_Index s             // starting node of the BFS
)
{
    GrB_Info info ;
    GrB_Index n ;                          // # of nodes in the graph
    GrB_Vector q = NULL ;                  // nodes visited at each level
    GrB_Vector v = NULL ;                  // result vector
    GrB_Descriptor desc = NULL ;           // Descriptor for mxv
    GrB_Matrix_nrows (&n, A) ;             // n = # of rows of A
    GrB_Vector_new (&v, GrB_INT32, n) ;    // Vector<int32_t> v(n) = 0
    for (int32_t i = 0 ; i < n ; i++)
        GrB_Vector_setElement (v, 0, i) ;
    GrB_Vector_new (&q, GrB_BOOL, n) ;     // Vector<bool> q(n) = false
    GrB_Vector_setElement (q, s, true) ;   // q[s] = true, false elsewhere
    GrB_Descriptor_new (&desc) ;
    GrB_Descriptor_set (desc, GrB_MASK, GrB_SCMP) ;     // invert the mask
    GrB_Descriptor_set (desc, GrB_OUTP, GrB_REPLACE) ;  // clear q first

    bool successor = true ; // true when some successor found
    for (int32_t level = 1 ; successor && level <= n ; level++)
    {
        // v<q> = level, using vector assign with q as the mask
        GrB_assign (v, q, NULL, level, GrB_ALL, n, NULL) ;
        // q<!v> = A ||.&& q ; finds all the unvisited
        // successors from current q, using !v as the mask
        GrB_mxv (q, v, NULL, GrB_LOR_LAND_BOOL, A, q, desc) ;
        // successor = ||(q)
        GrB_reduce (&successor, NULL, GrB_LOR_BOOL_MONOID, q, NULL) ;
    }
    *v_output = v ;         // return result
    GrB_free (&q) ;
    GrB_free (&desc) ;
    return (GrB_SUCCESS) ;
}
```

Fig. 1. GraphBLAS breadth-first search: given an adjacency matrix A and a source node s, perform a BFS traversal of the graph, setting v[i] to the level in which node i is visited.

```
v = zeros (1,n) ;       % v(k) is the BFS level (1 for source node s)
q = false (1,n) ;       % boolean vector of size n
q (s) = true ;          % q(k) is true if node k is in current level
for level = 1:n
    v (q) = level ;     % set v(i)=level where q(i) is true
    % new q = all unvisited neighbors of current q:
    t = A*q ;           % where '*' is the OR-AND semiring
    q = false (1,n) ;   % clear q of all entries
    q (~v) = t ;        % q (i) = t (i) but only where v(i) is zero
    if (~any (q)) break ;
end
```

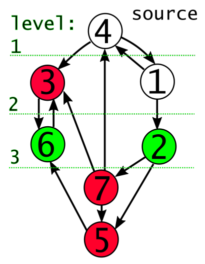

Fig. 2. Breadth-first search using pseudo-MATLAB notation. The only aspect of this code fragment that is not pure MATLAB is the computation of A*q, since MATLAB does provide matrix multiplication with the OR-AND semiring. The graphic shows the computation of t=A*q using the OR-AND semiring, where q is at level three (nodes 2 and 6). Level 4 will be nodes 5 and 7; node 3 is excluded since v is used as a complemented mask and v[3] is nonzero.

5.2. Luby's maximal independent set algorithm

A second algorithm provided with SuiteSparse:GraphBLAS is an implementation of Luby's parallel method for finding a maximal independent set [Luby 1986]. SuiteSparse:GraphBLAS is currently a sequential package, but it expresses the algorithm using whole-graph operations that could be implemented in parallel in future versions.

The *maximal independent set* problem is to find a set of nodes S such that no two nodes in S are adjacent to each other (an independent set), and all nodes not in S are adjacent to at least one node in S (and thus S is maximal since it cannot be augmented by any node while remaining an independent set).

In each phase, all candidate nodes are given a random score. If a node has a score higher than all its neighbors, then it is added to the independent set. All new nodes added to the set cause their neighbors to be removed from the set of candidates. The process must be repeated for multiple phases until no new nodes can be added. This is because in one phase, a node `i` might not be added because one of its neighbors `j` has a higher score, yet that neighbor `j` might not be added because one of its neighbors `k` is added to the independent set instead. The node `j` is no longer a candidate and can never be added to the independent set, but node `i` could be added to S in a subsequent phase.

Each phase of Luby's algorithm consists of nine calls to GraphBLAS operations. The inner loop of Luby's method is shown in Figure 3. The descriptor `r_desc` causes the result to be cleared first, and `sr_desc` selects that option in addition to complementing the mask.

The two matrix-vector multiplications are the important parts and also take the most time. They also make interesting use of semirings and masks. The first one discussed here computes the largest score of all the neighbors of each node in the candidate set:

```
// compute the max probability of all neighbors
GrB_mxv (neighbor_max, candidates, NULL, GxB_MAX_SECOND_FP32, A, prob, r_desc) ;
```

`A` is a Boolean matrix and `prob` is a sparse real vector of type `FP32`, which is a `float` in C. `prob(j)` is nonzero only if node `j` is a candidate. The pre-defined `GxB_MAX_SECOND_FP32` semiring uses `z=SECOND(x,y)=y` as the "multiply" operator. The row `A(i,:)` is the adjacency of node `i`, and the dot product `A(i,:)*prob` applies the `SECOND` operator on all entries that appear in the intersection of `A(i,:)` and `prob`, `z=SECOND(A(i,j),prob(j))` which is just `prob(j)` if `A(i,j)` is present. If `A(i,j)` is not an explicit entry in the matrix, then this term is not computed and does not take part in the reduction by the `MAX` monoid.

Thus, each term `z=SECOND(A(i,j),prob(j))` is the score, `prob(j)`, of all neighbors `j` of node `i` that have a score. Node `j` does not have a score if it is not also a candidate and so this is skipped. These terms are then "summed" up by taking the maximum score, using `MAX` as the additive monoid.

Finally, the results of this matrix-vector multiply are written to the result, `neighbor_max`. The `r_desc` descriptor has the `REPLACE` option enabled. Since `neighbor_max` does not also take part in the computation `A*prob`, it is simply cleared first. Next, is it modified only in those positions `i` where `candidates(i)` is true, using `candidates` as a mask. This sets the `neighbor_max` only for candidate nodes, and leaves the other components of `neighbor_max` as zero (implicit values not in the pattern of the vector).

All of the above work is done in a single matrix-vector multiply, with an elegant use of the max-second semiring coupled with a mask. The matrix-vector multiplication is described above as if it uses dot products of rows of `A` with the column vector `prob`, but SuiteSparse:GraphBLAS does not compute it that way. Sparse dot products are much

```
// Iterate while there are candidates to check.
GrB_Index nvals ;
GrB_Vector_nvals (&nvals, candidates) ;
int64_t last_nvals = nvals ;

while (nvals > 0)
{
    // compute a random probability scaled by inverse of degree
    GrB_apply (prob, candidates, NULL, set_random, degrees, r_desc) ;

    // compute the max probability of all neighbors
    GrB_mxv (neighbor_max, candidates, NULL, GxB_MAX_SECOND_FP32, A, prob, r_desc) ;

    // select node if its probability is > than all its active neighbors
    GrB_eWiseAdd (new_members, NULL, NULL, GrB_GT_FP64, prob, neighbor_max, NULL) ;

    // add new members to independent set.
    GrB_eWiseAdd (iset, NULL, NULL, GrB_LOR, iset, new_members, NULL) ;

    // remove new members from set of candidates c = c & !new
    GrB_apply (candidates, new_members, NULL, GrB_IDENTITY_BOOL, candidates, sr_desc) ;

    GrB_Vector_nvals (&nvals, candidates) ;
    if (nvals == 0) { break ; }                  // early exit condition

    // Neighbors of new members can also be removed from candidates
    GrB_mxv (new_neighbors, candidates, NULL, GxB_LOR_LAND_BOOL, A, new_members, NULL) ;
    GrB_apply (candidates, new_neighbors, NULL, GrB_IDENTITY_BOOL, candidates, sr_desc) ;

    GrB_Vector_nvals (&nvals, candidates) ;

    // this will not occur, unless the input is corrupted somehow
    if (last_nvals == nvals) { printf ("stall!\n") ; exit (1) ; }
    last_nvals = nvals ;
}
```

Fig. 3. Luby's maximal independent set method in GraphBLAS

slower than the optimal method for multiplying a sparse matrix times a sparse vector. The result is the same, however.

6. PERFORMANCE

GraphBLAS allows a user application to express its graph algorithms in a powerful, expressive, and simple manner. Details of the graph data structure and the implementation of the basic methods can be left to the author of the GraphBLAS library. The key question, however, is whether or not performance of such an application can match that of a graph algorithm written without GraphBLAS.

In [Davis 2018], two algorithms were written in GraphBLAS, and without GraphBLAS (both sequentially and in parallel). The first algorithm is to count the number of triangles in a graph, which is the number of unique cliques of size 3 in the graph. The second is construct the k-truss of a graph. In a 3-truss, all edges in the graph are removed except for those in one or more triangles. In a k-truss, each edge that remains must appear in at least $k-2$ triangles. Both codes are very simple to write using GraphBLAS.

The methods were tested on a large set of graphs from the MIT GraphChallenge (https://graphchallenge.mit.edu). Considering just the sequential implementation, the results show that triangle counting in SuiteSparse:GraphBLAS is competitive with a highly optimized single-threaded method, and even faster for some larger graphs. An asymptotically optimal yet basic C implementation of triangle counting is much slower than an implementation using SuiteSparse:GraphBLAS.

K-truss in SuiteSparse:GraphBLAS is also simple and the performance is competitive with a highly optimized and complex algorithm in pure C, rarely taking more than twice the time as the highly-optimized, sequential versions in pure C, and the GraphBLAS implementations are sometimes faster. These results demonstrate that GraphBLAS can be an efficient library that allows end users to write simple yet fast code.

Full details of this experiment are available in [Davis 2018], and code is available at http://suitesparse.com.

7. SUMMARY

The SuiteSparse:GraphBLAS library provides an efficient and highly optimized implementation of the GraphBLAS API standard [Buluç et al. 2017]. It allows users to write powerful and expressive graph algorithms with simple user-level code, whose performance is competitive with a highly-tuned code written by an expert. The package is available as a Collected Algorithm of the ACM, at http://suitesparse.com, and it is also available as a pre-packaged Debian and Ubuntu Linux distro in the suitesparse package.

REFERENCES

A. Azad, A. Buluç, and J. Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 804–811. DOI:http://dx.doi.org/10.1109/IPDPSW.2015.75

A. Buluç and J. R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IPDPS08: the IEEE International Symposium on Parallel & Distributed Processing*. IEEE Computer Society, 1–11.

A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. 2017. *The GraphBLAS C API Specification*. Technical Report. http://graphblas.org/.

T. A. Davis. 2006. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA.

T. A. Davis. 2018. Graph algorithms via SuiteSparse:GraphBLAS: triangle counting and K-truss. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*. (submitted; available at http://suitesparse.com).

T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566.

J. R. Gilbert, C. Moler, and R. Schreiber. 1992. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356. http://dx.doi.org/10.1137/0613024

F. G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978), 250–269. http://dx.doi.org/10.1145/355791.355796

J. Kepner. 2017. *GraphBLAS Mathematics*. Technical Report. http://www.mit.edu/∼kepner/GraphBLAS/GraphBLAS-Math-release.pdf.

J. Kepner and J. Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM, Philadelphia, PA.

M. Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* 15, 4 (1986). https://doi.org/10.1137/0215074.

Received June 2018; revised xxx 2018; accepted xxx 2018