The LAGraph User Guide Version 1.0 DRAFT

[Tim: Remember to update acknowledgements and remove DRAFT]

Tim Davis, Tim Mattson, Scott McMillan, and others from the LAGraph group who commit major blocks of time to write this thing

Generated on 2022/07/06 at 17:44:40 EDT

Contents

List of Tables

List of Figures

Acknowledgments

This document represents the work of the people who have served on the LAGraph Subcommittee of the GraphBLAS Forum.

Those who served as LAGraph API Subcommittee members are (in alphabetical order):

- David Bader (New Jersey Institute of Technology)
- Tim Davis (Texas A&M University)
- Jim Kitchen (Anaconda)
- Roi Lipman (redis Labs)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- Michel Pelletier (Graphegon Inc)
- Gabor Szarnyas (wherever)
- Erick Welch (Anaconda)

The LAGraph Library is based upon work funded and supported in part by:

- Intel Corporation
- Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute [DM-0003727, DM19-0929, DM21-0090]
- Graphegon Inc.
- Anaconda

The following people provided valuable input and feedback during the development of the LAGraph library (in alphabetical order): Benjamin Brock, Aydın Buluç, José Moreira

Chapter 1

Introduction

General introduction to LAGraph and its dependence on the GraphBLAS. We need to explain the motivation as well.

Normative standards include GraphBLAS version 2.0 and C99 (ISO/IEC 9899:199) extended with *static type-based* and *number of parameters-based* function polymorphism, and language extensions on par with the `_Generic` construct from C11 (ISO/IEC 9899:2011). Furthermore, the standard assumes programs using the LAGraph Library will execute on hardware that supports floating point arithmetic such as that defined by the IEEE 754 (IEEE 754-2008) standard.

Some more overview text to set the context for what follows

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects and defined values
- Chapter 4: The LAGraph API
- Appendix A: Revision history
- Appendix B: Examples

Chapter 2

Basic concepts

The LAGraph library is a collection of high level graph algorithms based on the GraphBLAS C API. These algorithms construct graph algorithms expressed "in the language of linear algebra." Graphs are expressed as matrices, and the operations over these matrices are generalized through the use of a semiring algebraic structure.

In this chapter, we will define the basic concepts used to define the LAGraph Library We provide the following elements:

- Glossary of terms and notation used in this document.
- The LAGraph objects.
- Return codes and other constants used in LAGraph.

Currently, I've kept the text from the GraphBLAS concepts chapter in this document. We may want to borrow some of the GraphBLAS glossary items and perhaps use some of the table formatting in LAGraph.

2.1 Glossary

2.1.1 LAGraph basic definitions

- *application*: A program that calls methods from the GraphBLAS C API to solve a problem.
- *GraphBLAS C API*: The application programming interface that fully defines the types, objects, literals, and other elements of the C binding to the GraphBLAS.
- *function*: Refers to a named group of statements in the C programming language. Methods, operators, and user-defined functions are typically implemented as C functions. When referring to the code programmers write, as opposed to the role of functions as an element of the GraphBLAS, they may be referred to as such.

- *method*: A function defined in the GraphBLAS C API that manipulates GraphBLAS objects or other opaque features of the implementation of the GraphBLAS API.

- *operator*: A function that performs an operation on the elements stored in GraphBLAS matrices and vectors.

- *GraphBLAS operation*: A mathematical operation defined in the GraphBLAS mathematical specification. These operations (not to be confused with *operators*) typically act on matrices and vectors with elements defined in terms of an algebraic semiring.

2.1.2 LAGraph objects and their structure

- *non-opaque datatype*: Any datatype that exposes its internal structure and can be manipulated directly by the user.

- *opaque datatype*: Any datatype that hides its internal structure and can be manipulated only through an API.

- *GraphBLAS object*: An instance of an *opaque datatype* defined by the *GraphBLAS C API* that is manipulated only through the GraphBLAS API. There are four kinds of GraphBLAS opaque objects: *domains* (i.e., types), *algebraic objects* (operators, monoids and semirings), *collections* (scalars, vectors, matrices and masks), and descriptors.

- *handle*: A variable that holds a reference to an instance of one of the GraphBLAS opaque objects. The value of this variable holds a reference to a GraphBLAS object but not the contents of the object itself. Hence, assigning a value to another variable copies the reference to the GraphBLAS object of one handle but not the contents of the object.

- *domain*: The set of valid values for the elements stored in a GraphBLAS *collection* or operated on by a GraphBLAS *operator*. Note that some GraphBLAS objects involve functions that map values from one or more input domains onto values in an output domain. These GraphBLAS objects would have multiple domains.

- *collection*: An opaque GraphBLAS object that holds a number of elements from a specified *domain*. Because these objects are based on an opaque datatype, an implementation of the GraphBLAS C API has the flexibility to optimize the data structures for a particular platform. GraphBLAS objects are often implemented as sparse data structures, meaning only the subset of the elements that have values are stored.

- *implied zero*: Any element that has a valid index (or indices) in a GraphBLAS vector or matrix but is not explicitly identified in the list of elements of that vector or matrix. From a mathematical perspective, an *implied zero* is treated as having the value of the zero element of the relevant monoid or semiring. However, GraphBLAS operations are purposefully defined using set notation in such a way that it makes it unnecessary to reason about implied zeros. Therefore, this concept is not used in the definition of GraphBLAS methods and operators.

- *mask*: An internal GraphBLAS object used to control how values are stored in a method's output object. The mask exists only inside a method; hence, it is called an *internal opaque* *object*. A mask is formed from the elements of a collection object (vector or matrix) input as a mask parameter to a method. GraphBLAS allows two types of masks:

1. In the default case, an element of the mask exists for each element that exists in the input collection object when the value of that element, when cast to a Boolean type, evaluates to `true`.
2. In the *structure only* case, masks have structure but no values. The input collection describes a structure whereby an element of the mask exists for each element stored in the input collection regardless of its value.

- *complement*: The *complement* of a GraphBLAS mask, M, is another mask, M', where the elements of M' are those elements from M that *do not* exist.

2.1.3 Algebraic structures used in the GraphBLAS

- *associative operator*: In an expression where a binary operator is used two or more times consecutively, that operator is *associative* if the result does not change regardless of the way operations are grouped (without changing their order). In other words, in a sequence of binary operations using the same associative operator, the legal placement of parenthesis does not change the value resulting from the sequence operations. Operators that are associative over infinitely precise numbers (e.g., real numbers) are not strictly associative when applied to numbers with finite precision (e.g., floating point numbers). Such non-associativity results, for example, from roundoff errors or from the fact some numbers can not be represented exactly as floating point numbers. In the GraphBLAS specification, as is common practice in computing, we refer to operators as *associative* when their mathematical definition over infinitely precise numbers is associative even when they are only approximately associative when applied to finite precision numbers.

  No GraphBLAS method will imply a predefined grouping over any associative operators. Implementations of the GraphBLAS are encouraged to exploit associativity to optimize performance of any GraphBLAS method with this requirement. This holds even if the definition of the GraphBLAS method implies a fixed order for the associative operations.

- *commutative operator*: In an expression where a binary operator is used (usually two or more times consecutively), that operator is *commutative* if the result does not change regardless of the order the inputs are operated on.

  No GraphBLAS method will imply a predefined ordering over any commutative operators. Implementations of the GraphBLAS are encouraged to exploit commutativity to optimize performance of any GraphBLAS method with this requirement. This holds even if the definition of the GraphBLAS method implies a fixed order for the commutative operations.

- *GraphBLAS operators*: Binary or unary operators that act on elements of GraphBLAS objects. *GraphBLAS operators* are used to express algebraic structures used in the GraphBLAS such as monoids and semirings. They are also used as arguments to several GraphBLAS methods. There are two types of *GraphBLAS operators*: (1) predefined operators found in Table 3.5 and (2) user-defined operators created using GrB_UnaryOp_new() or GrB_BinaryOp_new().

- *monoid*: An algebraic structure consisting of one domain, an associative binary operator, and the identity of that operator. There are two types of GraphBLAS monoids: (1) predefined monoids found in Table 3.7 and (2) user-defined monoids created using .

- *semiring*: An algebraic structure consisting of a set of allowed values (the *domain*), a commutative and associative binary operator called addition, a binary operator called multiplication (where multiplication distributes over addition), and identities over addition (*0*) and multiplication (*1*). The additive identity is an annihilator over multiplication.

- *GraphBLAS semiring*: is allowed to diverge from the mathematically rigorous definition of a *semiring* since certain combinations of domains, operators, and identity elements are useful in graph algorithms even when they do not strictly match the mathematical definition of a semiring. There are two types of *GraphBLAS semirings*: (1) predefined semirings found in Tables 3.8 and 3.9, and (2) user-defined semirings created using GrB_Semiring_new() (see Section **??**).

- *index unary operator*: A variation of the unary operator that operates on elements of GraphBLAS vectors and matrices along with the index values representing their location in the objects. There are predefined index unary operators found in Table 3.6), and user-defined operators created using GrB_IndexUnaryOp_new (see Section **??**).

2.1.4 The execution of an application using the GraphBLAS C API

- *program order*: The order of the GraphBLAS method calls in a thread, as defined by the text of the program.

- *host programming environment*: The GraphBLAS specification defines an API. The functions from the API appear in a program. This program is written using a programming language and execution environment defined outside of the GraphBLAS. We refer to this programming environment as the "host programming environment".

- *execution time*: time expended while executing instructions defined by a program. This term is specifically used in this specification in the context of computations carried out on behalf of a call to a GraphBLAS method.

- *sequence*: A GraphBLAS application uniquely defines a directed acyclic graph (DAG) of GraphBLAS method calls based on their program order. At any point in a program, the state of any GraphBLAS object is defined by a subgraph of that DAG. An ordered collection of GraphBLAS method calls in program order that defines that subgraph for a particular object is the *sequence* for that object.

- *complete*: A GraphBLAS object is complete when it can be used in a happens-before relationship with a method call that reads the variable on another thread. This concept is used when reasoning about memory orders in multithreaded programs. A GraphBLAS object defined on one thread that is complete can be safely used as an IN or INOUT argument in a method-call on a second thread assuming the method calls are correctly synchronized so the definition on the first thread *happens-before* it is used on the second thread. In blocking-mode, an object is complete after a GraphBLAS method call that writes to that object returns. In nonblocking-mode, an object is complete after a call to the GrB_wait() method with the GrB_COMPLETE parameter.

- *materialize*: A GraphBLAS object is materialized when it is (1) complete, (2) the computations defined by the sequence that define the object have finished (either fully or stopped at an error) and will not consume any additional computational resources, and (3) any errors associated with that sequence are available to be read according to the GraphBLAS error model. A GraphBLAS object that is never loaded into a non-opaque data structure may potentially never be materialized. This might happen, for example, if the operations associated with the object are fused or otherwise changed by the runtime system that supports the implementation of the GraphBLAS C API. An object can be materialized by a call to the materialize mode of the GrB_wait() method.

- *context*: An instance of the GraphBLAS C API implementation as seen by an application. An application can have only one context between the start and end of the application. A context begins with the first thread that calls GrB_init() and ends with the first thread to call GrB_finalize(). It is an error for GrB_init() or GrB_finalize() to be called more than one time within an application. The context is used to constrain the behavior of an instance of the GraphBLAS C API implementation and support various execution strategies. Currently, the only supported constraints on a context pertain to the mode of program execution.

- *program execution mode*: Defines how a GraphBLAS sequence executes, and is associated with the *context* of a GraphBLAS C API implementation. It is set by an application with its call to GrB_init() to one of two possible states. In *blocking mode*, GraphBLAS methods return after the computations complete and any output objects have been materialized. In *nonblocking mode*, a method may return once the arguments are tested as consistent with the method (i.e., there are no API errors), and potentially before any computation has taken place.

2.1.5 GraphBLAS methods: behaviors and error conditions

- *implementation-defined behavior*: Behavior that must be documented by the implementation and is allowed to vary among different compliant implementations.

- *undefined behavior*: Behavior that is not specified by the GraphBLAS C API. A conforming implementation is free to choose results delivered from a method whose behavior is undefined.

- *thread-safe*: Consider a function called from multiple threads with arguments that do not overlap in memory (i.e. the argument lists do not share memory). If the function is *thread-safe* then it will behave the same when executed concurrently by multiple threads or sequentially on a single thread.

- *dimension compatible*: GraphBLAS objects (matrices and vectors) that are passed as parameters to a GraphBLAS method are dimension (or shape) compatible if they have the correct number of dimensions and sizes for each dimension to satisfy the rules of the mathematical definition of the operation associated with the method. If any *dimension compatibility* rule above is violated, execution of the GraphBLAS method ends and the GrB_DIMENSION_MISMATCH error is returned.

- *domain compatible*: Two domains for which values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other, and a domain from a user-defined type is only compatible with itself. If any *domain compatibility* rule above is violated, execution of the GraphBLAS method ends and the GrB_DOMAIN_MISMATCH error is returned.

299 ## 2.2 Notation

Notation	Description
$D_{out}, D_{in}, D_{in_1}, D_{in_2}$	Refers to output and input domains of various GraphBLAS operators.
$\mathbf{D}_{out}(*), \mathbf{D}_{in}(*), \mathbf{D}_{in_1}(*), \mathbf{D}_{in_2}(*)$	Evaluates to output and input domains of GraphBLAS operators (usually a unary or binary operator, or semiring).
$\mathbf{D}(*)$	Evaluates to the (only) domain of a GraphBLAS object (usually a monoid, vector, or matrix).
f	An arbitrary unary function, usually a component of a unary operator.
$\mathbf{f}(F_u)$	Evaluates to the unary function contained in the unary operator given as the argument.
$\odot$	An arbitrary binary function, usually a component of a binary operator.
$\bigodot(*)$	Evaluates to the binary function contained in the binary operator or monoid given as the argument.
$\otimes$	Multiplicative binary operator of a semiring.
$\oplus$	Additive binary operator of a semiring.
$\bigotimes(S)$	Evaluates to the multiplicative binary operator of the semiring given as the argument.
$\bigoplus(S)$	Evaluates to the additive binary operator of the semiring given as the argument.
$\mathbf{0}(*)$	The identity of a monoid, or the additive identity of a GraphBLAS semiring.
$\mathbf{L}(*)$	The contents (all stored values) of the vector or matrix GraphBLAS objects. For a vector, it is the set of (index, value) pairs, and for a matrix it is the set of (row, col, value) triples.
300 $\mathbf{v}(i)$ or v_i	The i^{th} element of the vector $\mathbf{v}$.
$\mathbf{size}(\mathbf{v})$	The size of the vector $\mathbf{v}$.
$\mathbf{ind}(\mathbf{v})$	The set of indices corresponding to the stored values of the vector $\mathbf{v}$.
$\mathbf{nrows}(\mathbf{A})$	The number of rows in the $\mathbf{A}$.
$\mathbf{ncols}(\mathbf{A})$	The number of columns in the $\mathbf{A}$.
$\mathbf{indrow}(\mathbf{A})$	The set of row indices corresponding to rows in $\mathbf{A}$ that have stored values.
$\mathbf{indcol}(\mathbf{A})$	The set of column indices corresponding to columns in $\mathbf{A}$ that have stored values.
$\mathbf{ind}(\mathbf{A})$	The set of (i, j) indices corresponding to the stored values of the matrix.
$\mathbf{A}(i, j)$ or A_{ij}	The element of $\mathbf{A}$ with row index i and column index j.
$\mathbf{A}(:, j)$	The j^{th} column of matrix $\mathbf{A}$.
$\mathbf{A}(i, :)$	The i^{th} row of matrix $\mathbf{A}$.
$\mathbf{A}^T$	The transpose of matrix $\mathbf{A}$.
$\neg\mathbf{M}$	The complement of $\mathbf{M}$.
$\mathrm{s}(\mathbf{M})$	The structure of $\mathbf{M}$.
$\widetilde{\mathbf{t}}$	A temporary object created by the GraphBLAS implementation.
$< type >$	A method argument type that is `void *` or one of the types from Table 3.2.
GrB_ALL	A method argument literal to indicate that all indices of an input array should be used.
GrB_Type	A method argument type that is either a user defined type or one of the types from Table 3.2.
GrB_Object	A method argument type referencing any of the GraphBLAS object types.
GrB_NULL	The GraphBLAS NULL.

2.3 Mathematical foundations

Graphs can be represented in terms of matrices. The values stored in these matrices correspond to attributes (often weights) of edges in the graph.[1] Likewise, information about vertices in a graph are stored in vectors. The set of valid values that can be stored in either matrices or vectors is referred to as their domain. Matrices are usually sparse because the lack of an edge between two vertices means that nothing is stored at the corresponding location in the matrix. Vectors may be sparse or dense, or they may start out sparse and become dense as algorithms traverse the graphs.

Operations defined by the GraphBLAS C API specification operate on these matrices and vectors to carry out graph algorithms. These GraphBLAS operations are defined in terms of GraphBLAS semiring algebraic structures. Modifying the underlying semiring changes the result of an operation to support a wide range of graph algorithms. Inside a given algorithm, it is often beneficial to change the GraphBLAS semiring that applies to an operation on a matrix. This has two implications for the C binding of the GraphBLAS API.

First, it means that we define a separate object for the semiring to pass into methods. Since in many cases the full semiring is not required, we also support passing monoids or even binary operators, which means the semiring is implied rather than explicitly stated.

Second, the ability to change semirings impacts the meaning of the *implied zero* in a sparse representation of a matrix or vector. This element in real arithmetic is zero, which is the identity of the *addition* operator and the annihilator of the *multiplication* operator. As the semiring changes, this implied zero changes to the identity of the *addition* operator and the annihilator (if present) of the *multiplication* operator for the new semiring. Nothing changes regarding what is stored in the sparse matrix or vector, but the implied zeros within them change with respect to a particular operation. In all cases, the nature of the implied zero does not matter since the GraphBLAS C API requires that implementations treat them as nonexistent elements of the matrix or vector.

As with matrices and vectors, GraphBLAS semirings have domains associated with their inputs and outputs. The semirings in the GraphBLAS C API are defined with two domains associated with the input operands and one domain associated with output. When used in the GraphBLAS C API these domains may not match the domains of the matrices and vectors supplied in the operations. In this case, only valid *domain compatible* casting is supported by the API.

The mathematical formalism for graph operations in the language of linear algebra often assumes that we can operate in the field of real numbers. However, the GraphBLAS C binding is designed for implementation on computers, which by necessity have a finite number of bits to represent numbers. Therefore, we require a conforming implementation to use floating point numbers such as those defined by the IEEE-754 standard (both single- and double-precision) wherever real numbers need to be represented. The practical implications of these finite precision numbers is that the result of a sequence of computations may vary from one execution to the next as the grouping of operands (because of associativity) within the operations changes. While techniques are known to reduce these effects, we do not require or even expect an implementation to use them as they may add

[1] More information on the mathematical foundations can be found in the following paper: J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, J. Moreira, J. Owens, C. Yang, M. Zalewski, and T. Mattson. 2016, September. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1-9). IEEE.

Table 2.1: Types of GraphBLAS opaque objects.

GrB_Object types	Description
GrB_Type	Scalar type.
GrB_UnaryOp	Unary operator.
GrB_IndexUnaryOp	Unary operator, that operates on a single value and its location index values.
GrB_BinaryOp	Binary operator.
GrB_Monoid	Monoid algebraic structure.
GrB_Semiring	A GraphBLAS semiring algebraic structure.
GrB_Scalar	One element; could be empty.
GrB_Vector	One-dimensional collection of elements; can be sparse.
GrB_Matrix	Two-dimensional collection of elements; typically sparse.
GrB_Descriptor	Descriptor object, used to modify behavior of methods (specifically GraphBLAS operations).

considerable overhead. In most cases, these roundoff errors are not significant. When they are significant, the problem itself is ill-conditioned and needs to be reformulated.

2.4 LAgraph objects

Objects defined in the GraphBLAS standard include types (the domains of elements), collections of elements (matrices, vectors, and scalars), operators on those elements (unary, index unary, and binary operators), algebraic structures (semirings and monoids), and descriptors. GraphBLAS objects are defined as opaque types; that is, they are managed, manipulated, and accessed solely through the GraphBLAS application programming interface. This gives an implementation of the GraphBLAS C specification flexibility to optimize objects for different scenarios or to meet the needs of different hardware platforms.

A GraphBLAS opaque object is accessed through its *handle*. A handle is a variable that references an instance of one of the types from Table 2.1. An implementation of the GraphBLAS specification has a great deal of flexibility in how these handles are implemented. All that is required is that the handle corresponds to a type defined in the C language that supports assignment and comparison for equality. The GraphBLAS specification defines a literal GrB_INVALID_HANDLE that is valid for each type. Using the logical equality operator from C, it must be possible to compare a handle to GrB_INVALID_HANDLE to verify that a handle is valid.

Every GraphBLAS object has a *lifetime*, which consists of the sequence of instructions executed in program order between the *creation* and the *destruction* of the object. The GraphBLAS C API predefines a number of these objects which are created when the GraphBLAS context is initialized by a call to GrB_init and are destroyed when the GraphBLAS context is terminated by a call to GrB_finalize.

An application using the GraphBLAS API can create additional objects by declaring variables of the appropriate type from Table 2.1 for the objects it will use. Before use, the object must be initialized with a call call to one of the object's respective *constructor* methods. Each kind of object has at least one explicit constructor method of the form GrB_*_new where '*' is replaced with the type of object (e.g., GrB_Semiring_new). Note that some objects, especially collections, have additional constructor methods such as duplication, import, or deserialization. Objects explicitly created by a call to a constructor should be destroyed by a call to GrB_free. The behavior of a program that calls GrB_free on a pre-defined object is undefined.

These constructor and destructor methods are the only methods that change the value of a handle. Hence, objects changed by these methods are passed into the method as pointers. In all other cases, handles are not changed by the method and are passed by value. For example, even when multiplying matrices, while the contents of the output product matrix changes, the handle for that matrix is unchanged.

Several GraphBLAS constructor methods take other objects as input arguments and use these objects to create a new object. For all these methods, the lifetime of the created object must end strictly before the lifetime of any dependent input objects. For example, a vector constructor GrB_Vector_new takes a GrB_Type object as input. That type object must not be destroyed until after the created vector is destroyed. Similarly, a GrB_Semiring_new method takes a monoid and a binary operator as inputs. Neither of these can be destroyed until after the created semiring is destroyed.

Note that some constructor methods like GrB_Vector_dup and GrB_Matrix_dup behave differently. In these cases, the input vector or matrix can be destroyed as soon as the call returns. However, the original type object used to create the input vector or matrix cannot be destroyed until after the vector or matrix created by GrB_Vector_dup or GrB_Matrix_dup is destroyed. This behavior must hold for any chain of duplicating constructors.

Programmers using GraphBLAS handles must be careful to distinguish between a handle and the object manipulated through a handle. For example, a program may declare two GraphBLAS objects of the same type, initialize one, and then assign it to the other variable. That assignment, however, only assigns the handle to the variable. It does not create a copy of that variable (to do that, one would need to use the appropriate duplication method). If later the object is freed by calling GrB_free with the first variable, the object is destroyed and the second variable is left referencing an object that no longer exists (a so-called "dangling handle").

In addition to opaque objects manipulated through handles, the GraphBLAS C API defines an additional opaque object as an internal object; that is, the object is never exposed as a variable within an application. This opaque object is the mask used to control which computed values can be stored in the output operand of a *GraphBLAS operation*. .

Chapter 3

Objects

In this chapter, all of the enumerations, literals, data types, and predefined opaque objects defined in the GraphBLAS API are presented. Enumeration literals in GraphBLAS are assigned specific values to ensure compatibility between different runtime library implementations. The chapter starts by defining the enumerations that are used by the init() and wait() methods. Then a number of transparent (i.e., non-opaque) types that are used for interfacing with external data are defined. Sections that follow describe the various types of opaque objects in GraphBLAS: types (or *domains*), algebraic objects, collections and descriptors. Each of these sections also lists the predefined instances of each opaque type that are required by the API. This chapter concludes with a section on the definition for GrB_Info enumeration that is used as the return type of all methods.

3.1 Enumerations for init() and wait()

Table 3.1 lists the enumerations and the corresponding values used in the GrB_init() method to set the execution mode and in the GrB_wait() method for completing or materializing opaque objects.

3.2 Indices, index arrays, and scalar arrays

In order to interface with third-party software (i.e., software other than an implementation of the GraphBLAS), operations such as GrB_Matrix_build (Section ??) and GrB_Matrix_extractTuples (Section ??) must specify how the data should be laid out in non-opaque data structures. To this end we explicitly define the types for indices and the arrays used by these operations.

For indices a typedef is used to give a GraphBLAS name to a concrete type. We define it as follows:

```
typedef uint64_t GrB_Index;
```

The range of valid values for a variable of type GrB_Index is [0, GrB_INDEX_MAX] where the largest index value permissible is defined with a macro, GrB_INDEX_MAX. For example:

```
#define GrB_INDEX_MAX ((GrB_Index) 0x0fffffffffffffff);
```

An implementation is required to define and document this value.

An index array is a pointer to a set of GrB_Index values that are stored in a contiguous block of memory (i.e., GrB_Index*). Likewise, a scalar array is a pointer to a contiguous block of memory storing a number of scalar values as specified by the user. Some GraphBLAS operations (e.g., GrB_assign) include an input parameter with the type of an index array. This input index array selects a subset of elements from a GraphBLAS vector or matrix object to be used in the operation. In these cases, the literal GrB_ALL can be used in place of the index array input parameter to indicate that all indices of the associated GraphBLAS vector or matrix object should be used. An implementation of the GraphBLAS C API has considerable freedom in terms of how GrB_ALL is defined. Since GrB_ALL is used as an argument for an array parameter, it must use a type consistent with a pointer. GrB_ALL must also have a non-null value to distinguish it from the erroneous case of passing a NULL pointer as an array.

3.3 Types (domains)

In GraphBLAS, domains correspond to the valid values for types from the host language (in our case, the C programming language). GraphBLAS defines a number of operators that take elements from one or more domains and produce elements of a (possibly) different domain. GraphBLAS also defines three kinds of collections: matrices, vectors and scalars. For any given collection, the elements of the collection belong to a *domain*, which is the set of valid values for the elements. For any variable or object V in GraphBLAS we denote as $\mathbf{D}(V)$ the domain of V, that is, the set of possible values that elements of V can take.

Table 3.1: Enumeration literals and corresponding values input to various GraphBLAS methods.

(a) GrB_Mode execution modes for the GrB_init method.

Symbol	Value	Description
GrB_NONBLOCKING	0	Specifies the nonblocking mode context.
GrB_BLOCKING	1	Specifies the blocking mode context.

(b) GrB_WaitMode wait modes for the GrB_wait method.

Symbol	Value	Description
GrB_COMPLETE	0	The object is in a state where it can be used in a happens-before relation so that multithreaded programs can be properly synchronized.
GrB_MATERIALIZE	1	The object is *complete*, and in addition, all computation of the object is finished and any error information is available.

Table 3.2: Predefined GrB_Type values, and the corresponding GraphBLAS domain suffixes, C type (for scalar parameters), and domains for GraphBLAS. The domain suffixes are used in place of I, F, and T in Tables 3.5, 3.6, 3.7, 3.8, and 3.9).

GrB_Type	Suffix	C type	Domain
GrB_BOOL	BOOL	`bool`	$\{\texttt{false}, \texttt{true}\}$
GrB_INT8	INT8	`int8_t`	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	UINT8	`uint8_t`	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	INT16	`int16_t`	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	UINT16	`uint16_t`	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	INT32	`int32_t`	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	UINT32	`uint32_t`	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	INT64	`int64_t`	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	UINT64	`uint64_t`	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	FP32	`float`	IEEE 754 binary32
GrB_FP64	FP64	`double`	IEEE 754 binary64

The domains for elements that can be stored in collections and operated on through GraphBLAS methods are defined by GraphBLAS objects called GrB_Type. The predefined types and corresponding domains used in the GraphBLAS C API are shown in Table 3.2. The Boolean type (`bool`) is defined in `stdbool.h`, the integral types (`int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`) are defined in `stdint.h`, and the floating-point types (`float`, `double`) are native to the language and platform and in most cases defined by the IEEE-754 standard.

3.4 Algebraic objects, operators and associated functions

GraphBLAS operators operate on elements stored in GraphBLAS collections. A *binary operator* is a function that maps two input values to one output value. A *unary operator* is a function that maps one input value to one output value. Binary operators are defined over two input domains and produce an output from a (possibly different) third domain. Unary operators are specified over one input domain and produce an output from a (possibly different) second domain.

In addition to the operators that operate on stored values, GraphBLAS also supports *index unary operators* that maps a stored value and the indices of its position in the matrix or vector to an output value. That output value can be used in the index unary operator variants of apply (§ **??**) to compute a new stored value, or be used in the select operation (§ **??**) to determine if the stored input value should be kept or annihilated.

Some GraphBLAS operations require a monoid or semiring. A monoid contains an associative binary operator where the input and output domains are the same. The monoid also includes an identity value of the operator. The semiring consists of a binary operator – referred to as the "times" operator – with up to three different domains (two inputs and one output) and a monoid

Table 3.3: Operator input for relevant GraphBLAS operations. The semiring add and times are shown if applicable.

Operation	Operator input
mxm, mxv, vxm	semiring
eWiseAdd	binary operator monoid semiring (add)
eWiseMult	binary operator monoid semiring (times)
reduce (to vector or GrB_Scalar)	binary operator monoid
reduce (to scalar value)	monoid
apply	unary operator binary operator with scalar index unary operator
select	index unary operator
kronecker	binary operator monoid semiring
dup argument (build methods)	binary operator
accum argument (various methods)	binary operator

– referred to as the "plus" operator – that is also commutative. Furthermore, the domain of the monoid must be the same as the output domain of the "times" operator.

The GraphBLAS *algebraic objects* operators, monoids, and semirings are presented in this section. These objects can be used as input arguments to various GraphBLAS operations, as shown in Table 3.3. The specific rules for each algebraic object are explained in the respective sections of those objects. A summary of the properties and recipes for building these GraphBLAS algebraic objects is presented in Table 3.4.

A number of predefined operators are specified by the GraphBLAS C API. They are presented in tables in their respective subsections below. Each of these operators is defined to operate on specific GraphBLAS types and therefore, this type is built into the name of the object as a suffix. These suffixes and the corresponding predefined GrB_Type objects that are listed in Table 3.2.

3.4.1 Operators

A GraphBLAS *unary operator* $F_u = \langle D_{out}, D_{in}, f \rangle$ is defined by two domains, D_{out} and D_{in}, and an operation $f : D_{in} \rightarrow D_{out}$. For a given GraphBLAS unary operator $F_u = \langle D_{out}, D_{in}, f \rangle$, we define $\mathbf{D}_{out}(F_u) = D_{out}$, $\mathbf{D}_{in}(F_u) = D_{in}$, and $\mathbf{f}(F_u) = f$.

A GraphBLAS *binary operator* $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$ is defined by three domains, D_{out}, D_{in_1},

Table 3.4: Properties and recipes for building GraphBLAS algebraic objects: unary operator, binary operator, monoid, and semiring (composed of operations *add* and *times*).

(a) Properties of algebraic objects.

Object	Must be commutative	Must be associative	Identity must exist	Number of domains
Unary operator	n/a	n/a	n/a	2
Binary operator	no	no	no	3
Monoid	no	yes	yes	1
Reduction add	yes	yes	yes (see Note 1)	1
Semiring add	yes	yes	yes	1
Semiring times	no	no	no	3 (see Note 2)

(b) Recipes for algebraic objects.

Object	Recipe	Number of domains
Unary operator	Function pointer	2
Binary operator	Function pointer	3
Monoid	Associative binary operator with identity	1
Semiring	Commutative monoid + binary operator	3

Note 1: Some high-performance GraphBLAS implementations may require an identity to perform reductions to sparse objects like GraphBLAS vectors and scalars. According to the descriptions of the corresponding GraphBLAS operations, however, this identity is mathematically not necessary. There are API signatures to support both.
Note 2: The output domain of the semiring times must be same as the domain of the semiring's add monoid. This ensures three domains for a semiring rather than four.

D_{in_2}, and an operation $\odot : D_{in_1} \times D_{in_2} \rightarrow D_{out}$. For a given GraphBLAS binary operator $F_b = \langle D_{out}, D_{in_1}, D_{in_2}, \odot \rangle$, we define $\mathbf{D}_{out}(F_b) = D_{out}$, $\mathbf{D}_{in_1}(F_b) = D_{in_1}$, $\mathbf{D}_{in_2}(F_b) = D_{in_2}$, and $\bigodot(F_b) = \odot$. Note that $\odot$ could be used in place of either $\oplus$ or $\otimes$ in other methods and operations.

A GraphBLAS *index unary operator* $F_i = \langle D_{out}, D_{in_1}, \mathbf{D}(\mathsf{GrB\_Index}), D_{in_2}, f_i \rangle$ is defined by three domains, D_{out}, D_{in_1}, D_{in_2}, the domain of GraphBLAS indices, and an operation $f_i : D_{in_1} \times I_{U64}^2 \times D_{in_2} \rightarrow D_{out}$ (where I_{U64} corresponds to the domain of a GrB_Index). For a given GraphBLAS index operator F_i, we define $\mathbf{D}_{out}(F_i) = D_{out}$, $\mathbf{D}_{in_1}(F_i) = D_{in_1}$, $\mathbf{D}_{in_2}(F_i) = D_{in_2}$, and $\mathbf{f}(F_i) = f_i$.

User-defined operators can be created with calls to GrB_UnaryOp_new, GrB_BinaryOp_new, and GrB_IndexUnaryOp_new, respectively. See Section **??** for information on these methods. The GraphBLAS C API predefines a number of these operators. These are listed in Tables 3.5 and 3.6. Note that most entries in these tables represent a "family" of predefined operators for a set of different types represented by the T, I, or F in their names. For example, the multiplicative inverse (GrB_MINV_F) function is only defined for floating-point types (F = FP32 or FP64). The division (GrB_DIV_T) function is defined for all types, but only if $y \neq 0$ for integral and floating point types and $y \neq$ `false` for the Boolean type.

Table 3.5: Predefined unary and binary operators for GraphBLAS in C. The T can be any suffix from Table 3.2, I can be any integer suffix from Table 3.2, and F can be any floating-point suffix from Table 3.2.

Operator type	GraphBLAS identifier	Domains	Description	
GrB_UnaryOp	GrB_IDENTITY_T	$T \rightarrow T$	$f(x) = x,$	identity
GrB_UnaryOp	GrB_ABS_T	$T \rightarrow T$	$f(x) = \|x\|,$	absolute value
GrB_UnaryOp	GrB_AINV_T	$T \rightarrow T$	$f(x) = -x,$	additive inverse
GrB_UnaryOp	GrB_MINV_F	$F \rightarrow F$	$f(x) = \frac{1}{x},$	multiplicative inverse
GrB_UnaryOp	GrB_LNOT	`bool` $\rightarrow$ `bool`	$f(x) = \neg x,$	logical inverse
GrB_UnaryOp	GrB_BNOT_I	$I \rightarrow I$	$f(x) = \tilde{\ } x,$	bitwise complement
GrB_BinaryOp	GrB_LOR	`bool` $\times$ `bool` $\rightarrow$ `bool`	$f(x, y) = x \vee y,$	logical OR
GrB_BinaryOp	GrB_LAND	`bool` $\times$ `bool` $\rightarrow$ `bool`	$f(x, y) = x \wedge y,$	logical AND
GrB_BinaryOp	GrB_LXOR	`bool` $\times$ `bool` $\rightarrow$ `bool`	$f(x, y) = x \oplus y,$	logical XOR
GrB_BinaryOp	GrB_LXNOR	`bool` $\times$ `bool` $\rightarrow$ `bool`	$f(x, y) = \overline{x \oplus y},$	logical XNOR
GrB_BinaryOp	GrB_BOR_I	$I \times I \rightarrow I$	$f(x, y) = x \mid y,$	bitwise OR
GrB_BinaryOp	GrB_BAND_I	$I \times I \rightarrow I$	$f(x, y) = x \ \& \ y,$	bitwise AND
GrB_BinaryOp	GrB_BXOR_I	$I \times I \rightarrow I$	$f(x, y) = x \ \hat{} \ y,$	bitwise XOR
GrB_BinaryOp	GrB_BXNOR_I	$I \times I \rightarrow I$	$f(x, y) = \overline{x \ \hat{} \ y},$	bitwise XNOR
GrB_BinaryOp	GrB_EQ_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x == y)$	equal
GrB_BinaryOp	GrB_NE_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x \neq y)$	not equal
GrB_BinaryOp	GrB_GT_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x > y)$	greater than
GrB_BinaryOp	GrB_LT_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x < y)$	less than
GrB_BinaryOp	GrB_GE_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x \geq y)$	greater than or equal
GrB_BinaryOp	GrB_LE_T	$T \times T \rightarrow$ `bool`	$f(x, y) = (x \leq y)$	less than or equal
GrB_BinaryOp	GrB_ONEB_T	$T \times T \rightarrow T$	$f(x, y) = 1,$	1 (cast to T)
GrB_BinaryOp	GrB_FIRST_T	$T \times T \rightarrow T$	$f(x, y) = x,$	first argument
GrB_BinaryOp	GrB_SECOND_T	$T \times T \rightarrow T$	$f(x, y) = y,$	second argument
GrB_BinaryOp	GrB_MIN_T	$T \times T \rightarrow T$	$f(x, y) = (x < y) \ ? \ x : y,$	minimum
GrB_BinaryOp	GrB_MAX_T	$T \times T \rightarrow T$	$f(x, y) = (x > y) \ ? \ x : y,$	maximum
GrB_BinaryOp	GrB_PLUS_T	$T \times T \rightarrow T$	$f(x, y) = x + y,$	addition
GrB_BinaryOp	GrB_MINUS_T	$T \times T \rightarrow T$	$f(x, y) = x - y,$	subtraction
GrB_BinaryOp	GrB_TIMES_T	$T \times T \rightarrow T$	$f(x, y) = xy,$	multiplication
GrB_BinaryOp	GrB_DIV_T	$T \times T \rightarrow T$	$f(x, y) = \frac{x}{y},$	division

Table 3.6: Predefined index unary operators for GraphBLAS in C. The T can be any suffix from Table 3.2. I_{U64} refers to the unsigned 64-bit, GrB_Index, integer type, I_{32} refers to the signed, 32-bit integer type, and I_{64} refers to signed, 64-bit integer type. The parameters, u_i or A_{ij}, are the stored values from the containers where the i and j parameters are set to the row and column indices corresponding to the location of the stored value. When operating on vectors, j will be passed with a zero value. Finally, s is an additional scalar value used in the operators. The expressions in the "Description" column are to be treated as mathematical specifications. That is, for the index arithmetic functions in the first two groups below, each one of i, j, and s is interpreted as an integer number in the set $\mathbb{Z}$. Functions are evaluated using arithmetic in $\mathbb{Z}$, producing a result value that is also in $\mathbb{Z}$. The result value is converted to the output type according to the rules of the C language. In particular, if the value cannot be represented as a signed 32- or 64-bit integer type, the output is implementation defined. Any deviations from this ideal behavior, including limitations on the values of i, j, and s, or possible overflow and underflow conditions, must be defined by the implementation.

Operator type Type	GraphBLAS Name	Domains (– is don't care) A, u	i, j	s	result	Description			
GrB_IndexUnaryOp	GrB_ROWINDEX_$I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s)$	=	$(i + s)$,	replace with its row index (+ s)
		–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(u_i, i, 0, s)$	=	$(i + s)$	
GrB_IndexUnaryOp	GrB_COLINDEX_$I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s)$	=	$(j + s)$	replace with its column index (+ s)
GrB_IndexUnaryOp	GrB_DIAGINDEX_$I_{32/64}$	–	I_{U64}	$I_{32/64}$	$I_{32/64}$	$f(A_{ij}, i, j, s)$	=	$(j - i + s)$	replace with its diagonal index (+ s)
GrB_IndexUnaryOp	GrB_TRIL	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j \leq i + s)$	triangle on or below diagonal s
GrB_IndexUnaryOp	GrB_TRIU	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j \geq i + s)$	triangle on or above diagonal s
GrB_IndexUnaryOp	GrB_DIAG	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j == i + s)$	diagonal s
GrB_IndexUnaryOp	GrB_OFFDIAG	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j \neq i + s)$	all but diagonal s
GrB_IndexUnaryOp	GrB_COLLE	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j \leq s)$	columns less or equal to s
GrB_IndexUnaryOp	GrB_COLGT	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(j > s)$	columns greater than s
GrB_IndexUnaryOp	GrB_ROWLE	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(i \leq s)$,	rows less or equal to s
		–	I_{U64}	I_{64}	bool	$f(u_i, i, 0, s)$	=	$(i \leq s)$	
GrB_IndexUnaryOp	GrB_ROWGT	–	I_{U64}	I_{64}	bool	$f(A_{ij}, i, j, s)$	=	$(i > s)$,	rows greater than s
		–	I_{U64}	I_{64}	bool	$f(u_i, i, 0, s)$	=	$(i > s)$	
GrB_IndexUnaryOp	GrB_VALUEEQ_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} == s)$,	elements equal to value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i == s)$	
GrB_IndexUnaryOp	GrB_VALUENE_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} \neq s)$,	elements not equal to value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i \neq s)$	
GrB_IndexUnaryOp	GrB_VALUELT_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} < s)$,	elements less than value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i < s)$	
GrB_IndexUnaryOp	GrB_VALUELE_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} \leq s)$,	elements less or equal to value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i \leq s)$	
GrB_IndexUnaryOp	GrB_VALUEGT_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} > s)$,	elements greater than value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i > s)$	
GrB_IndexUnaryOp	GrB_VALUEGE_T	T	–	T	bool	$f(A_{ij}, i, j, s)$	=	$(A_{ij} \geq s)$,	elements greater or equal to value s
		T	–	T	bool	$f(u_i, i, 0, s)$	=	$(u_i \geq s)$	

3.4.2 Monoids

A GraphBLAS *monoid* $M = \langle D, \odot, 0\rangle$ is defined by a single domain D, an *associative*[1] operation $\odot : D \times D \to D$, and an identity element $0 \in D$. For a given GraphBLAS monoid $M = \langle D, \odot, 0\rangle$ we define $\mathbf{D}(M) = D$, $\bigodot(M) = \odot$, and $\mathbf{0}(M) = 0$. A GraphBLAS monoid is equivalent to the conventional *monoid* algebraic structure.

Let $F = \langle D, D, D, \odot\rangle$ be an associative GraphBLAS binary operator with identity element $0 \in D$. Then $M = \langle F, 0\rangle = \langle D, \odot, 0\rangle$ is a GraphBLAS monoid. If $\odot$ is commutative, then M is said to be a *commutative monoid*. If a monoid M is created using an operator $\odot$ that is not associative, the outcome of GraphBLAS operations using such a monoid is undefined.

User-defined monoids can be created with calls to GrB_Monoid_new (see Section **??**). The GraphBLAS C API predefines a number of monoids that are listed in Table 3.7. Predefined monoids are named GrB_*op*_MONOID_T, where *op* is the name of the predefined GraphBLAS operator used as the associative binary operation of the monoid and T is the domain (type) of the monoid.

3.4.3 Semirings

A GraphBLAS *semiring* $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ is defined by three domains D_{out}, D_{in_1}, and D_{in_2}; an *associative*[1] and commutative additive operation $\oplus : D_{out} \times D_{out} \to D_{out}$; a multiplicative operation $\otimes : D_{in_1} \times D_{in_2} \to D_{out}$; and an identity element $0 \in D_{out}$. For a given GraphBLAS semiring $S = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ we define $\mathbf{D}_{in_1}(S) = D_{in_1}$, $\mathbf{D}_{in_2}(S) = D_{in_2}$, $\mathbf{D}_{out}(S) = D_{out}$, $\bigoplus(S) = \oplus$, $\bigotimes(S) = \otimes$, and $\mathbf{0}(S) = 0$.

Let $F = \langle D_{out}, D_{in_1}, D_{in_2}, \otimes\rangle$ be an operator and let $A = \langle D_{out}, \oplus, 0\rangle$ be a commutative monoid, then $S = \langle A, F\rangle = \langle D_{out}, D_{in_1}, D_{in_2}, \oplus, \otimes, 0\rangle$ is a semiring.

In a GraphBLAS semiring, the multiplicative operator does not have to distribute over the additive operator. This is unlike the conventional *semiring* algebraic structure.

Note: There must be one GraphBLAS monoid in every semiring which serves as the semiring's additive operator and specifies the same domain for its inputs and output parameters. If this monoid is not a commutative monoid, the outcome of GraphBLAS operations using the semiring is undefined.

User-defined semirings can be created with calls to GrB_Semiring_new (see Section **??**). A list of predefined true semirings and convenience semirings can be found in Tables 3.8 and 3.9, respectively. Predefined semirings are named GrB_*add*_*mul*_SEMIRING_T, where *add* is the semiring additive operation, *mul* is the semiring multiplicative operation and T is the domain (type) of the semiring.

[1] It is expected that implementations of the GraphBLAS will utilize floating point arithmetic such as that defined in the IEEE-754 standard even though floating point arithmetic is not strictly associative.

Table 3.7: Predefined monoids for GraphBLAS in C. Maximum and minimum values for the various integral types are defined in `stdint.h`. Floating-point infinities are defined in `math.h`. The x in UINTx or INTx can be one of 8, 16, 32, or 64; whereas in FPx, it can be 32 or 64.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	Identity	Description
GrB_PLUS_MONOID_T	UINTx	0	addition
	INTx	0	
	FPx	0	
GrB_TIMES_MONOID_T	UINTx	1	multiplication
	INTx	1	
	FPx	1	
GrB_MIN_MONOID_T	UINTx	UINTx_MAX	minimum
	INTx	INTx_MAX	
	FPx	INFINITY	
GrB_MAX_MONOID_T	UINTx	0	maximum
	INTx	INTx_MIN	
	FPx	-INFINITY	
GrB_LOR_MONOID_BOOL	BOOL	false	logical OR
GrB_LAND_MONOID_BOOL	BOOL	true	logical AND
GrB_LXOR_MONOID_BOOL	BOOL	false	logical XOR (not equal)
GrB_LXNOR_MONOID_BOOL	BOOL	true	logical XNOR (equal)

Table 3.8: Predefined true semirings for GraphBLAS in C where the additive identity is the multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in UINTx or INTx, and can be 32 or 64 in FPx.

GraphBLAS identifier	Domains, T ($T \times T \rightarrow T$)	$+$ identity $\times$ annihilator	Description
GrB_PLUS_TIMES_SEMIRING_T	UINTx	0	arithmetic semiring
	INTx	0	
	FPx	0	
GrB_MIN_PLUS_SEMIRING_T	UINTx	UINTx_MAX	min-plus semiring
	INTx	INTx_MAX	
	FPx	INFINITY	
GrB_MAX_PLUS_SEMIRING_T	INTx	INTx_MIN	max-plus semiring
	FPx	-INFINITY	
GrB_MIN_TIMES_SEMIRING_T	UINTx	UINTx_MAX	min-times semiring
GrB_MIN_MAX_SEMIRING_T	UINTx	UINTx_MAX	min-max semiring
	INTx	INTx_MAX	
	FPx	INFINITY	
GrB_MAX_MIN_SEMIRING_T	UINTx	0	max-min semiring
	INTx	INTx_MIN	
	FPx	-INFINITY	
GrB_MAX_TIMES_SEMIRING_T	UINTx	0	max-times semiring
GrB_PLUS_MIN_SEMIRING_T	UINTx	0	plus-min semiring
GrB_LOR_LAND_SEMIRING_BOOL	BOOL	false	Logical semiring
GrB_LAND_LOR_SEMIRING_BOOL	BOOL	true	"and-or" semiring
GrB_LXOR_LAND_SEMIRING_BOOL	BOOL	false	same as NE_LAND
GrB_LXNOR_LOR_SEMIRING_BOOL	BOOL	true	same as EQ_LOR

Table 3.9: Other useful predefined semirings for GraphBLAS in C that don't have a multiplicative annihilator. The x can be one of 8, 16, 32, or 64 in UINTx or INTx, and can be 32 or 64 in FPx.

GraphBLAS identifier	Domains, T ($T \times T \to T$)	+ identity	Description
GrB_MAX_PLUS_SEMIRING_T	UINTx	0	max-plus semiring
GrB_MIN_TIMES_SEMIRING_T	INTx	INTx_MAX	min-times semiring
	FPx	INFINITY	
GrB_MAX_TIMES_SEMIRING_T	INTx	INTx_MIN	max-times semiring
	FPx	-INFINITY	
GrB_PLUS_MIN_SEMIRING_T	INTx	0	plus-min semiring
	FPx	0	
GrB_MIN_FIRST_SEMIRING_T	UINTx	UINTx_MAX	min-select first semiring
	INTx	INTx_MAX	
	FPx	INFINITY	
GrB_MIN_SECOND_SEMIRING_T	UINTx	UINTx_MAX	min-select second semiring
	INTx	INTx_MAX	
	FPx	INFINITY	
GrB_MAX_FIRST_SEMIRING_T	UINTx	0	max-select first semiring
	INTx	INTx_MIN	
	FPx	-INFINITY	
GrB_MAX_SECOND_SEMIRING_T	UINTx	0	max-select second semiring
	INTx	INTx_MIN	
	FPx	-INFINITY	

3.5 Collections

3.5.1 Scalars

A *GraphBLAS scalar*, $s = \langle D, \{\sigma\}\rangle$, is defined by a domain D, and a set of zero or one *scalar value*, σ, where $\sigma \in D$. We define $\mathbf{size}(s) = 1$ (constant), and $\mathbf{L}(s) = \{\sigma\}$. The set $\mathbf{L}(s)$ is called the *contents* of the GraphBLAS scalar s. We also define $\mathbf{D}(s) = D$. Finally, $\mathbf{val}(s)$ is a reference to the scalar value, σ, if the GraphBLAS scalar is not empty, and is undefined otherwise.

3.5.2 Vectors

A vector $\mathbf{v} = \langle D, N, \{(i, v_i)\}\rangle$ is defined by a domain D, a size $N > 0$, and a set of tuples (i, v_i) where $0 \leq i < N$ and $v_i \in D$. A particular value of i can appear at most once in $\mathbf{v}$. We define $\mathbf{size}(\mathbf{v}) = N$ and $\mathbf{L}(\mathbf{v}) = \{(i, v_i)\}$. The set $\mathbf{L}(\mathbf{v})$ is called the *content* of vector $\mathbf{v}$. We also define the set $\mathbf{ind}(\mathbf{v}) = \{i : (i, v_i) \in \mathbf{L}(\mathbf{v})\}$ (called the *structure* of $\mathbf{v}$), and $\mathbf{D}(\mathbf{v}) = D$. For a vector $\mathbf{v}$, $\mathbf{v}(i)$ is a reference to v_i if $(i, v_i) \in \mathbf{L}(\mathbf{v})$ and is undefined otherwise.

3.5.3 Matrices

A matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$ is defined by a domain D, its number of rows $M > 0$, its number of columns $N > 0$, and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and $A_{ij} \in D$. A particular pair of values i, j can appear at most once in $\mathbf{A}$. We define $\mathbf{ncols}(\mathbf{A}) = N$, $\mathbf{nrows}(\mathbf{A}) = M$, and $\mathbf{L}(\mathbf{A}) = \{(i, j, A_{ij})\}$. The set $\mathbf{L}(\mathbf{A})$ is called the *content* of matrix $\mathbf{A}$. We also define the sets $\mathbf{indrow}(\mathbf{A}) = \{i : \exists(i, j, A_{ij}) \in \mathbf{A}\}$ and $\mathbf{indcol}(\mathbf{A}) = \{j : \exists(i, j, A_{ij}) \in \mathbf{A}\}$. (These are the sets of nonempty rows and columns of $\mathbf{A}$, respectively.) The *structure* of matrix $\mathbf{A}$ is the set $\mathbf{ind}(\mathbf{A}) = \{(i, j) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}$, and $\mathbf{D}(\mathbf{A}) = D$. For a matrix $\mathbf{A}$, $\mathbf{A}(i, j)$ is a reference to A_{ij} if $(i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})$ and is undefined otherwise.

If $\mathbf{A}$ is a matrix and $0 \leq j < N$, then $\mathbf{A}(:, j) = \langle D, M, \{(i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$ is a vector called the j-th *column* of $\mathbf{A}$. Correspondingly, if $\mathbf{A}$ is a matrix and $0 \leq i < M$, then $\mathbf{A}(i, :) = \langle D, N, \{(j, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$ is a vector called the i-th *row* of $\mathbf{A}$.

Given a matrix $\mathbf{A} = \langle D, M, N, \{(i, j, A_{ij})\}\rangle$, its *transpose* is another matrix $\mathbf{A}^T = \langle D, N, M, \{(j, i, A_{ij}) : (i, j, A_{ij}) \in \mathbf{L}(\mathbf{A})\}\rangle$.

3.5.3.1 External matrix formats

The specification also supports the export and import of matrices to/from a number of commonly used formats, such as COO, CSR, and CSC formats. When importing or exporting a matrix to or from a GraphBLAS object using GrB_Matrix_import (§ **??**) or GrB_Matrix_export (§ **??**), it is necessary to specify the data format for the matrix data external to GraphBLAS, which is being imported from or exported to. This non-opaque data format is specified using an argument of enumeration type GrB_Format that is used to indicate one of a number of predefined formats. The predefined values of GrB_Format are specified in Table 3.10. A precise definition of the non-opaque data formats can be found in Appendix **??**.

Table 3.10: GrB_Format enumeration literals and corresponding values for matrix import and export methods.

Symbol	Value	Description
GrB_CSR_FORMAT	0	Specifies the compressed sparse row matrix format.
GrB_CSC_FORMAT	1	Specifies the compressed sparse column matrix format.
GrB_COO_FORMAT	2	Specifies the sparse coordinate matrix format.

3.5.4 Masks

The GraphBLAS C API defines an opaque object called a *mask*. The mask is used to control how computed values are stored in the output from a method. The mask is an *internal* opaque object; that is, it is never exposed as a variable within an application.

The mask is formed from input objects to the method that uses the mask. For example, a GraphBLAS method may be called with a matrix as the mask parameter. The internal mask object is constructed from the input matrix in one of two ways. In the default case, an element of the mask is created for each tuple that exists in the matrix for which the value of the tuple cast to Boolean evaluates to `true`. Alternatively, the user can specify *structure*-only behavior where an element of the mask is created for each tuple that exists in the matrix *regardless* of the value stored in the input matrix.

The internal mask object can be either a one- or a two-dimensional construct. One- and two-dimensional masks, described more formally below, are similar to vectors and matrices, respectively, except that they have structure (indices) but no values. When needed, a value is implied for the elements of a mask with an implied value of `true` for elements that exist and an implied value of `false` for elements that do not exist (i.e., the locations of the mask that do not have a stored value imply a value of `false`). Hence, even though a mask does not contain any values, it can be considered to imply values from a Boolean domain.

A one-dimensional mask $\mathbf{m} = \langle N, \{i\} \rangle$ is defined by its number of elements $N > 0$, and a set $\mathbf{ind}(\mathbf{m})$ of indices $\{i\}$ where $0 \leq i < N$. A particular value of i can appear at most once in $\mathbf{m}$. We define $\mathbf{size}(\mathbf{m}) = N$. The set $\mathbf{ind}(\mathbf{m})$ is called the *structure* of mask $\mathbf{m}$.

A two-dimensional mask $\mathbf{M} = \langle M, N, \{(i,j)\} \rangle$ is defined by its number of rows $M > 0$, its number of columns $N > 0$, and a set $\mathbf{ind}(\mathbf{M})$ of tuples (i,j) where $0 \leq i < M$, $0 \leq j < N$. A particular pair of values i, j can appear at most once in $\mathbf{M}$. We define $\mathbf{ncols}(\mathbf{M}) = N$, and $\mathbf{nrows}(\mathbf{M}) = M$. We also define the sets $\mathbf{indrow}(\mathbf{M}) = \{i : \exists(i,j) \in \mathbf{ind}(\mathbf{M})\}$ and $\mathbf{indcol}(\mathbf{M}) = \{j : \exists(i,j) \in \mathbf{ind}(\mathbf{M})\}$. These are the sets of nonempty rows and columns of $\mathbf{M}$, respectively. The set $\mathbf{ind}(\mathbf{M})$ is called the *structure* of mask $\mathbf{M}$.

One common operation on masks is the *complement*. For a one-dimensional mask $\mathbf{m}$ this is denoted as $\neg\mathbf{m}$. For a two-dimensional mask $\mathbf{M}$, this is denoted as $\neg\mathbf{M}$. The complement of a one-dimensional mask $\mathbf{m}$ is defined as $\mathbf{ind}(\neg\mathbf{m}) = \{i : 0 \leq i < N, i \notin \mathbf{ind}(\mathbf{m})\}$. It is the set of all possible indices that do not appear in $\mathbf{m}$. The complement of a two-dimensional mask $\mathbf{M}$ is defined as the set $\mathbf{ind}(\neg\mathbf{M}) = \{(i,j) : 0 \leq i < M, 0 \leq j < N, (i,j) \notin \mathbf{ind}(\mathbf{M})\}$. It is the set of all possible indices that do not appear in $\mathbf{M}$.

3.6 Descriptors

Descriptors are used to modify the behavior of a GraphBLAS method. When present in the signature of a method, they appear as the last argument in the method. Descriptors specify how the other input arguments corresponding to GraphBLAS collections – vectors, matrices, and masks – should be processed (modified) before the main operation of a method is performed. A complete list of what descriptors are capable of are presented in this section.

The descriptor is a lightweight object. It is composed of (*field*, *value*) pairs where the *field* selects one of the GraphBLAS objects from the argument list of a method and the *value* defines the indicated modification associated with that object. For example, a descriptor may specify that a particular input matrix needs to be transposed or that a mask needs to be complemented (defined in Section 3.5.4) before using it in the operation.

For the purpose of constructing descriptors, the arguments of a method that can be modified are identified by specific field names. The output parameter (typically the first parameter in a GraphBLAS method) is indicated by the field name, GrB_OUTP. The mask is indicated by the GrB_MASK field name. The input parameters corresponding to the input vectors and matrices are indicated by GrB_INP0 and GrB_INP1 in the order they appear in the signature of the GraphBLAS method. The descriptor is an opaque object and hence we do not define how objects of this type should be implemented. When referring to (*field*, *value*) pairs for a descriptor, however, we often use the informal notation desc[GrB_Desc_Field].GrB_Desc_Value without implying that a descriptor is to be implemented as an array of structures (in fact, field values can be used in conjunction with multiple values that are composable). We summarize all types, field names, and values used with descriptors in Table 3.11.

In the definitions of the GraphBLAS methods, we often refer to the *default behavior* of a method with respect to the action of a descriptor. If a descriptor is not provided or if the value associated with a particular field in a descriptor is not set, the default behavior of a GraphBLAS method is defined as follows:

- Input matrices are not transposed.
- The mask is used, as is, without complementing, and stored values are examined to determine whether they evaluate to `true` or `false`.
- Values of the output object that are not directly modified by the operation are preserved.

GraphBLAS specifies all of the valid combinations of (field, value) pairs as predefined descriptors. Their identifiers and the corresponding set of (field, value) pairs for that identfier are shown in Table 3.12.

Table 3.11: Descriptors are GraphBLAS objects passed as arguments to GraphBLAS operations to modify other GraphBLAS objects in the operation's argument list. A descriptor, desc, has one or more (*field*, *value*) pairs indicated as desc[GrB_Desc_Field].GrB_Desc_Value. In this table, we define all types and literals used with descriptors.

(a) Types used with GraphBLAS descriptors.

Type	Description
GrB_Descriptor	Type of a GraphBLAS descriptor object.
GrB_Desc_Field	The descriptor field enumeration.
GrB_Desc_Value	The descriptor value enumeration.

(b) Descriptor field names of type GrB_Desc_Field enumeration and corresponding values.

Field Name	Value	Description
GrB_OUTP	0	Field name for the output GraphBLAS object.
GrB_MASK	1	Field name for the mask GraphBLAS object.
GrB_INP0	2	Field name for the first input GraphBLAS object.
GrB_INP1	3	Field name for the second input GraphBLAS object.

(c) Descriptor field values of type GrB_Desc_Value enumeration and corresponding values.

Value Name	Value	Description
(reserved)	0	Unused
GrB_REPLACE	1	Clear the output object before assigning computed values.
GrB_COMP	2	Use the complement of the associated object. When combined with GrB_STRUCTURE, the complement of the structure of the associated object is used without evaluating the values stored.
GrB_TRAN	3	Use the transpose of the associated object.
GrB_STRUCTURE	4	The write mask is constructed from the structure (pattern of stored values) of the associated object. The stored values are not examined.

Table 3.12: Predefined GraphBLAS descriptors. The list includes all possible descriptors, according to the current standard. Columns list the possible fields and entries list the value(s) associated with those fields for a given descriptor.

Identifier	GrB_OUTP	GrB_MASK	GrB_INP0	GrB_INP1
GrB_NULL	–	–	–	–
GrB_DESC_T1	–	–	–	GrB_TRAN
GrB_DESC_T0	–	–	GrB_TRAN	–
GrB_DESC_T0T1	–	–	GrB_TRAN	GrB_TRAN
GrB_DESC_C	–	GrB_COMP	–	–
GrB_DESC_S	–	GrB_STRUCTURE	–	–
GrB_DESC_CT1	–	GrB_COMP	–	GrB_TRAN
GrB_DESC_ST1	–	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_CT0	–	GrB_COMP	GrB_TRAN	–
GrB_DESC_ST0	–	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_CT0T1	–	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_ST0T1	–	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_SC	–	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_SCT1	–	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_SCT0	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_SCT0T1	–	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_R	GrB_REPLACE	–	–	–
GrB_DESC_RT1	GrB_REPLACE	–	–	GrB_TRAN
GrB_DESC_RT0	GrB_REPLACE	–	GrB_TRAN	–
GrB_DESC_RT0T1	GrB_REPLACE	–	GrB_TRAN	GrB_TRAN
GrB_DESC_RC	GrB_REPLACE	GrB_COMP	–	–
GrB_DESC_RS	GrB_REPLACE	GrB_STRUCTURE	–	–
GrB_DESC_RCT1	GrB_REPLACE	GrB_COMP	–	GrB_TRAN
GrB_DESC_RST1	GrB_REPLACE	GrB_STRUCTURE	–	GrB_TRAN
GrB_DESC_RCT0	GrB_REPLACE	GrB_COMP	GrB_TRAN	–
GrB_DESC_RST0	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	–
GrB_DESC_RCT0T1	GrB_REPLACE	GrB_COMP	GrB_TRAN	GrB_TRAN
GrB_DESC_RST0T1	GrB_REPLACE	GrB_STRUCTURE	GrB_TRAN	GrB_TRAN
GrB_DESC_RSC	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	–
GrB_DESC_RSCT1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	–	GrB_TRAN
GrB_DESC_RSCT0	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	–
GrB_DESC_RSCT0T1	GrB_REPLACE	GrB_STRUCTURE, GrB_COMP	GrB_TRAN	GrB_TRAN

3.7 GrB_Info return values

All GraphBLAS methods return a GrB_Info enumeration value. The three types of return codes (informational, API error, and execution error) and their corresponding values are listed in Table 3.13.

Table 3.13: Enumeration literals and corresponding values returned by GraphBLAS methods and operations.

(a) Informational return values

Symbol	Value	Description
GrB_SUCCESS	0	The method/operation completed successfully (blocking mode), or encountered no API errors (non-blocking mode).
GrB_NO_VALUE	1	A location in a matrix or vector is being accessed that has no stored value at the specified location.

(b) API errors

Symbol	Value	Description
GrB_UNINITIALIZED_OBJECT	-1	A GraphBLAS object is passed to a method before new was called on it.
GrB_NULL_POINTER	-2	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	-3	Miscellaneous incorrect values.
GrB_INVALID_INDEX	-4	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	-5	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	-6	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	-7	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NOT_IMPLEMENTED	-8	An attempt was made to call a GraphBLAS method for a combination of input parameters that is not supported by a particular implementation.

(c) Execution errors

Symbol	Value	Description
GrB_PANIC	-101	Unknown internal error.
GrB_OUT_OF_MEMORY	-102	Not enough memory for operations.
GrB_INSUFFICIENT_SPACE	-103	The array provided is not large enough to hold output.
GrB_INVALID_OBJECT	-104	One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error.
GrB_INDEX_OUT_OF_BOUNDS	-105	Reference to a vector or matrix element that is outside the defined dimensions of the object.
GrB_EMPTY_OBJECT	-106	One of the opaque GraphBLAS objects does not have a stored value.

Chapter 4

LAGraph API

This chapter defines the behavior of all the functions in the LAGraph library. All methods can be declared for use in programs by including the `LAGraph.h` header file.

4.1 LAGraph_ConnectedComponents

Finds the connected components of an undirected graph.

C Syntax

```
int LAGr_ConnectedComponents
(
      GrB_Vector *component,
      LAGraph_Graph G,
      char *msg
 )
```

Parameters

*component (OUT) An array holding identifiers to the components.

G (IN) the input Graph (not modified by this function).

- msg A message meaning something.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully.

Either way, output matrix C is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or Matrix_dup for matrix parameters).

GrB_DIMENSION_MISMATCH Mask and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

Description

GrB_mxm computes the matrix product $\mathsf{C} = \mathsf{A} \oplus.\otimes \mathsf{B}$ or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{C} = \mathsf{C} \odot (\mathsf{A} \oplus . \otimes \mathsf{B})$ (where matrices A and B can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal matrices and mask used in the computation are formed and their domains and dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output matrix, possibly under control of a mask.

Up to four argument matrices are used in the GrB_mxm operation:

1. $\mathsf{C} = \langle \mathbf{D}(\mathsf{C}), \mathbf{nrows}(\mathsf{C}), \mathbf{ncols}(\mathsf{C}), \mathbf{L}(\mathsf{C}) = \{(i, j, C_{ij})\}\rangle$
2. $\mathsf{Mask} = \langle \mathbf{D}(\mathsf{Mask}), \mathbf{nrows}(\mathsf{Mask}), \mathbf{ncols}(\mathsf{Mask}), \mathbf{L}(\mathsf{Mask}) = \{(i, j, M_{ij})\}\rangle$ (optional)
3. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$
4. $\mathsf{B} = \langle \mathbf{D}(\mathsf{B}), \mathbf{nrows}(\mathsf{B}), \mathbf{ncols}(\mathsf{B}), \mathbf{L}(\mathsf{B}) = \{(i, j, B_{ij})\}\rangle$

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the matrix multiplication and any additional associated operations. We describe this in terms of two intermediate matrices:

- $\widetilde{\mathbf{T}}$: The matrix holding the product of matrices $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{B}}$.
- $\widetilde{\mathbf{Z}}$: The matrix holding the result after application of the (optional) accumulation operator.

The intermediate matrix $\widetilde{\mathbf{T}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{nrows}(\widetilde{\mathbf{A}}), \mathbf{ncols}(\widetilde{\mathbf{B}}), \{(i,j,T_{ij}) : \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:,j)) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$T_{ij} = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{A}}(i,:)) \cap \mathbf{ind}(\widetilde{\mathbf{B}}(:,j))} (\widetilde{\mathbf{A}}(i,k) \otimes \widetilde{\mathbf{B}}(k,j)),$$

where $\oplus$ and $\otimes$ are the additive and multiplicative operators of semiring op, respectively.

4.1.1 vxm: Vector-matrix multiply

Multiplies a (row) vector with a matrix on an semiring. The result is a vector.

C Syntax

```
GrB_Info GrB_vxm(GrB_Vector             w,
                 const GrB_Vector       mask,
                 const GrB_BinaryOp     accum,
                 const GrB_Semiring     op,
                 const GrB_Vector       u,
                 const GrB_Matrix       A,
                 const GrB_Descriptor   desc);
```

Parameters

w (INOUT) An existing GraphBLAS vector. On input, the vector provides values that may be accumulated with the result of the vector-matrix product. On output, this vector holds the results of the operation.

mask (IN) An optional "write" mask that controls which results from this operation are stored into the output vector w. The mask dimensions must match those of the vector w. If the GrB_STRUCTURE descriptor is *not* set for the mask, the domain of the mask vector must be of type bool or any of the predefined "built-in" types in Table 3.2. If the default mask is desired (i.e., a mask that is all true with the dimensions of w), GrB_NULL should be specified.

accum (IN) An optional binary operator used for accumulating entries into existing w entries. If assignment rather than accumulation is desired, GrB_NULL should be specified.

op (IN) Semiring used in the vector-matrix multiply.

u (IN) The GraphBLAS vector holding the values for the left-hand vector in the multiplication.

A (IN) The GraphBLAS matrix holding the values for the right-hand matrix in the multiplication.

desc (IN) An optional operation descriptor. If a *default* descriptor is desired, GrB_NULL should be specified. Non-default field/value pairs are listed as follows:

Param	Field	Value	Description
w	GrB_OUTP	GrB_REPLACE	Output vector w is cleared (all elements removed) before the result is stored in it.
mask	GrB_MASK	GrB_STRUCTURE	The write mask is constructed from the structure (pattern of stored values) of the input mask vector. The stored values are not examined.
mask	GrB_MASK	GrB_COMP	Use the complement of mask.
A	GrB_INP1	GrB_TRAN	Use transpose of A for the operation.

Return Values

GrB_SUCCESS In blocking mode, the operation completed successfully. In non-blocking mode, this indicates that the compatibility tests on dimensions and domains for the input arguments passed successfully. Either way, output vector w is ready to be used in the next method of the sequence.

GrB_PANIC Unknown internal error.

GrB_INVALID_OBJECT This is returned in any execution mode whenever one of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a previous execution error. Call GrB_error() to access any error messages generated by the implementation.

GrB_OUT_OF_MEMORY Not enough memory available for the operation.

GrB_UNINITIALIZED_OBJECT One or more of the GraphBLAS objects has not been initialized by a call to new (or dup for matrix or vector parameters).

GrB_DIMENSION_MISMATCH Mask, vector, and/or matrix dimensions are incompatible.

GrB_DOMAIN_MISMATCH The domains of the various vectors/matrices are incompatible with the corresponding domains of the semiring or accumulation operator, or the mask's domain is not compatible with bool (in the case where desc[GrB_MASK].GrB_STRUCTURE is not set).

Description

GrB_vxm computes the vector-matrix product $\mathsf{w}^T = \mathsf{u}^T \oplus . \otimes \mathsf{A}$, or, if an optional binary accumulation operator ($\odot$) is provided, $\mathsf{w}^T = \mathsf{w}^T \odot \left(\mathsf{u}^T \oplus . \otimes \mathsf{A}\right)$ (where matrix A can be optionally transposed). Logically, this operation occurs in three steps:

Setup The internal vectors, matrices and mask used in the computation are formed and their domains/dimensions are tested for compatibility.

Compute The indicated computations are carried out.

Output The result is written into the output vector, possibly under control of a mask.

Up to four argument vectors or matrices are used in the GrB_vxm operation:

1. $\mathsf{w} = \langle \mathbf{D}(\mathsf{w}), \mathbf{size}(\mathsf{w}), \mathbf{L}(\mathsf{w}) = \{(i, w_i)\}\rangle$
2. $\mathsf{mask} = \langle \mathbf{D}(\mathsf{mask}), \mathbf{size}(\mathsf{mask}), \mathbf{L}(\mathsf{mask}) = \{(i, m_i)\}\rangle$ (optional)
3. $\mathsf{u} = \langle \mathbf{D}(\mathsf{u}), \mathbf{size}(\mathsf{u}), \mathbf{L}(\mathsf{u}) = \{(i, u_i)\}\rangle$
4. $\mathsf{A} = \langle \mathbf{D}(\mathsf{A}), \mathbf{nrows}(\mathsf{A}), \mathbf{ncols}(\mathsf{A}), \mathbf{L}(\mathsf{A}) = \{(i, j, A_{ij})\}\rangle$

The argument matrices, vectors, the semiring, and the accumulation operator (if provided) are tested for domain compatibility as follows:

1. If mask is not GrB_NULL, and desc[GrB_MASK].GrB_STRUCTURE is not set, then $\mathbf{D}(\mathsf{mask})$ must be from one of the pre-defined types of Table 3.2.
2. $\mathbf{D}(\mathsf{u})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{op})$ of the semiring.
3. $\mathbf{D}(\mathsf{A})$ must be compatible with $\mathbf{D}_{in_2}(\mathsf{op})$ of the semiring.
4. $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{out}(\mathsf{op})$ of the semiring.
5. If accum is not GrB_NULL, then $\mathbf{D}(\mathsf{w})$ must be compatible with $\mathbf{D}_{in_1}(\mathsf{accum})$ and $\mathbf{D}_{out}(\mathsf{accum})$ of the accumulation operator and $\mathbf{D}_{out}(\mathsf{op})$ of the semiring must be compatible with $\mathbf{D}_{in_2}(\mathsf{accum})$ of the accumulation operator.

Two domains are compatible with each other if values from one domain can be cast to values in the other domain as per the rules of the C language. In particular, domains from Table 3.2 are all compatible with each other. A domain from a user-defined type is only compatible with itself. If any compatibility rule above is violated, execution of GrB_vxm ends and the domain mismatch error listed above is returned.

From the argument vectors and matrices, the internal matrices and mask used in the computation are formed ($\leftarrow$ denotes copy):

1. Vector $\widetilde{\mathbf{w}} \leftarrow \mathsf{w}$.
2. One-dimensional mask, $\widetilde{\mathbf{m}}$, is computed from argument mask as follows:
   (a) If $\mathsf{mask} = \mathsf{GrB\_NULL}$, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{w}), \{i,\ \forall\ i : 0 \leq i < \mathbf{size}(\mathsf{w})\}\rangle$.

(b) If mask $\neq$ GrB_NULL,

i. If desc[GrB_MASK].GrB_STRUCTURE is set, then $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask})\}\rangle$,

ii. Otherwise, $\widetilde{\mathbf{m}} = \langle \mathbf{size}(\mathsf{mask}), \{i : i \in \mathbf{ind}(\mathsf{mask}) \wedge (\mathsf{bool})\mathsf{mask}(i) = \mathsf{true}\}\rangle$.

(c) If desc[GrB_MASK].GrB_COMP is set, then $\widetilde{\mathbf{m}} \leftarrow \neg\widetilde{\mathbf{m}}$.

3. Vector $\widetilde{\mathbf{u}} \leftarrow \mathsf{u}$.

4. Matrix $\widetilde{\mathbf{A}} \leftarrow$ desc[GrB_INP1].GrB_TRAN ? A^T : A.

The internal matrices and masks are checked for shape compatibility. The following conditions must hold:

1. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{size}(\widetilde{\mathbf{m}})$.

2. $\mathbf{size}(\widetilde{\mathbf{w}}) = \mathbf{ncols}(\widetilde{\mathbf{A}})$.

3. $\mathbf{size}(\widetilde{\mathbf{u}}) = \mathbf{nrows}(\widetilde{\mathbf{A}})$.

If any compatibility rule above is violated, execution of GrB_vxm ends and the dimension mismatch error listed above is returned.

From this point forward, in GrB_NONBLOCKING mode, the method can optionally exit with GrB_SUCCESS return code and defer any computation and/or execution error codes.

We are now ready to carry out the vector-matrix multiplication and any additional associated operations. We describe this in terms of two intermediate vectors:

- $\widetilde{\mathbf{t}}$: The vector holding the product of vector $\widetilde{\mathbf{u}}^T$ and matrix $\widetilde{\mathbf{A}}$.

- $\widetilde{\mathbf{z}}$: The vector holding the result after application of the (optional) accumulation operator.

The intermediate vector $\widetilde{\mathbf{t}} = \langle \mathbf{D}_{out}(\mathsf{op}), \mathbf{ncols}(\widetilde{\mathbf{A}}), \{(j, t_j) : \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:, j)) \neq \emptyset\}\rangle$ is created. The value of each of its elements is computed by

$$t_j = \bigoplus_{k \in \mathbf{ind}(\widetilde{\mathbf{u}}) \cap \mathbf{ind}(\widetilde{\mathbf{A}}(:,j))} (\widetilde{\mathbf{u}}(k) \otimes \widetilde{\mathbf{A}}(k, j)),$$

where $\oplus$ and $\otimes$ are the additive and multiplicative operators of semiring op, respectively.

Appendix A

Revision history

This document defines the LAGraph 1.0 release and hence one could argue that there should not be a revision history just yet. Early pre-release versions of LAGraph, however, have been heavily used. We therefore need to summarize the key changes from the pre-release version of LAGraph and the official, 1.0 release. Changes in 1.0 (Released: 12 September 2022):

- We did a global redefinition of return codes to be more consistent and to mesh better with the GraphBLAS return codes.
- In the pre-release LAGraph library, we included type information on the LAGraph graph object. We have deprecated this feature since it is safer to use the type introspection from GraphBLAS than to carry distinct type information inside the LAGraph object.

Appendix B

Examples

Text to introduce the examples.

B.1 Example: Compute the page rank of a graph using LAGraph.

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "LAGraph.h"

void test_PageRank(void)
{
    LAGraph_Init (msg) ;
    GrB_Matrix A = NULL ;
    GrB_Vector centrality = NULL, cmatlab = NULL, diff = NULL ;
    int niters = 0 ;

    // create the karate graph
    snprintf (filename, LEN, LG_DATA_DIR "%s", "karate.mtx") ;
    FILE *f = fopen (filename, "r") ;
    TEST_CHECK (f != NULL) ;
    OK (LAGraph_MMRead (&A, f, msg)) ;
    OK (fclose (f)) ;
    OK (LAGraph_New (&G, &A, LAGraph_ADJACENCY_UNDIRECTED, msg)) ;
    TEST_CHECK (A == NULL) ;    // A has been moved into G->A
    OK (LAGraph_Cached_OutDegree (G, msg)) ;

    // compute its pagerank
    OK (LAGr_PageRank (&centrality, &niters, G, 0.85, 1e-4, 100, msg)) ;
    OK (LAGraph_Delete (&G, msg)) ;

    // compare with MATLAB: cmatlab = centrality (G, 'pagerank')
    float err = difference (centrality, karate_rank) ;
    printf ("\nkarate:   err: %e\n", err) ;
    TEST_CHECK (err < 1e-4) ;
    OK (GrB_free (&centrality)) ;

    LAGraph_Finalize (msg) ;
}
```

B.2 Example: Apply betweenness centrality algorithm to a Graph using LAGraph

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "LAGraph.h"

void test_bc (void)
{
    LAGraph_Init (msg) ;
    GrB_Matrix A = NULL ;
    GrB_Vector centrality = NULL ;
    int niters = 0 ;

    // create the karate graph
    snprintf (filename, LEN, LG_DATA_DIR "%s", "karate.mtx") ;
    FILE *f = fopen (filename, "r") ;
    TEST_CHECK (f != NULL) ;
    OK (LAGraph_MMRead (&A, f, msg)) ;
    OK (fclose (f)) ;
    OK (LAGraph_New (&G, &A, LAGraph_ADJACENCY_UNDIRECTED, msg)) ;
    TEST_CHECK (A == NULL) ;    // A has been moved into G->A

    // compute its betweenness centrality
    OK (LAGr_Betweenness (&centrality, G, karate_sources, 4, msg)) ;
    printf ("\nkarate bc:\n") ;
    OK (LAGraph_Delete (&G, msg)) ;

    // compare with GAP:
    float err = difference (centrality, karate_bc) ;
    printf ("karate:   err: %e\n", err) ;
    TEST_CHECK (err < 1e-4) ;
    OK (GrB_free (&centrality)) ;

    LAGraph_Finalize (msg) ;
}
```