The LAGraph C API Specification Version 0.5

David Bader, Tim Davis, Jim Kitchen, Scott Kolodziej, Tim Mattson, Scott McMillan, and others as they take on writing tasks

Generated on 2020/04/08 at 09:00:53 PDT

Contents

List of Tables

List of Figures

Acknowledgments

This document represents the work of the people who have served on the C API LAGraph subcommittee of the GraphBLAS Forum.

Those who served as C API subcommittee members for LAGraph 0.5 are (in alphabetical order):

- David Bader (New Jersey Institute of Technology)
- Aydın Buluç (Lawrence Berkeley National Laboratory)
- Tim Davis (Texas A&M)
- James Kitchen (Anaconda)
- Scott Kolodziej (Texas A&M)
- Timothy G. Mattson (Intel Corporation)
- Scott McMillan (Software Engineering Institute at Carnegie Mellon University)
- José Moreira (IBM Corporation)
- Gabor Szarnyas (???)

The following people provided valuable input and feedback during the development of the specification (in alphabetical order): *to be added later.*

Chapter 1

Introduction

The LAGraph C API defines a library of graph algorithms based on a representation of graphs in terms of linear algebra [2]. We assume an implementation of the LAGraph library is layered on top of a GraphBLAS library [1] and the objects manipulated by LAGraph are constructed from GraphBLAS objects.

LAGraph is built on a collection of objects exposed to the C programmer as opaque data types. Functions that manipulate these objects are referred to as *methods*. These methods fully define the interface to LAGraph objects to create or destroy them, modify their contents, and copy the contents of opaque objects into non-opaque objects; the contents of which are under direct control of the programmer.

The LAGraph C API is designed to work with 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).

The remainder of this document is organized as follows:

- Chapter 2: Basic Concepts
- Chapter 3: Objects
- Chapter 4: Functions
- Appendix A: Revision History
- Appendix B: Examples

Chapter 2

Basic Concepts

The GraphBLAS C API is used to 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 GraphBLAS C API. We provide the following elements:

- Glossary of terms used in this document.
- Notation
- Execution model
- Error model

2.1 Glossary

2.1.1 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.

2.1.2 Objects and their structure

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

- *non-opaque datatype*: Any datatype that exposes its internal structure. This is contrasted with an *opaque datatype* that hides its internal structure and can be manipulated only through an API.

105 ## 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.
106 $\mathbf{v}(i)$ 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 the the matrix $\mathbf{A}$.
$\mathbf{A}(i, :)$	The i^{th} row of the the matrix $\mathbf{A}$.
$\mathbf{A}^T$	The transpose of the matrix $\mathbf{A}$.
$\neg\mathbf{M}$	The complement of $\mathbf{M}$.
$\widetilde{\mathbf{t}}$	A temporary object created by the GraphBLAS implementation.
$< \mathit{type} >$	A method argument type that is void * or one of the types from Table **??**.
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 **??**.
GrB_Object	A method argument type referencing any of the GraphBLAS object types.
GrB_NULL	The GraphBLAS NULL.

2.3 Error model

2.4 Execution Model

A program using the GraphBLAS C API constructs GraphBLAS objects, manipulates them to implement a graph algorithm, and then extracts values from the GraphBLAS objects as the result of the algorithm. Functions defined within the GraphBLAS C API that manipulate GraphBLAS objects are called *methods*. If the method corresponds to one of the operations defined in the GraphBLAS mathematical specification, we refer to the method as an *operation*.

Graph algorithms are expressed as an ordered collection of GraphBLAS method calls defined by the order they are encountered in a program. This is called the *program order*. Each method in the collection uniquely and unambiguously defines the output GraphBLAS objects based on the GraphBLAS operation and the input GraphBLAS objects. This is the case as long as there are no execution errors, which can put objects in an invalid state (see Section 2.5).

The GraphBLAS method calls in program order are organized into contiguous and nonoverlapping *sequences*. A sequence is an ordered collection of method calls as encountered by an executing thread. (For more on threads and GraphBLAS, see Section **??**.) A sequence begins with either (1) the first GraphBLAS method called by a thread, or (2) the first method called by a thread after the end of the previous sequence. A sequence can end (terminate) in a variety of ways. A call to the GraphBLAS GrB_wait() method (Section **??**) always ends a sequence. The GraphBLAS GrB_finalize() method (Section **??**) also implicitly ends a sequence. Finally, in blocking mode (see below), each GraphBLAS method starts and ends its own sequence.

The GraphBLAS objects are fully defined at any point in a sequence by the methods in the sequence as long as there are no execution errors. In particular, as soon as a GraphBLAS method call returns, its output can be used in the next GraphBLAS method call. However, individual operations in a sequence may not be *complete*. We say that an operation is complete when all the computations in the operation have finished and all the values of its output object have been produced and committed to the address space of the program. Furthermore, no additional execution time can be charged to a completed operation and no additional errors can be attributed to a completed operation.

The opaqueness of GraphBLAS objects allows execution to proceed from one method to the next even when operations are not complete. Processing of nonopaque objects is never deferred in GraphBLAS. That is, methods that consume nonopaque objects (e.g., GrB_Matrix_build(), Section **??**) and methods that produce nonopaque objects (e.g., GrB_Matrix_extractTuples(), Section **??**) always finish consuming or producing those nonopaque objects before returning.

2.5 Error Model

All GraphBLAS methods return a value of type GrB_Info to provide information available to the system at the time the method returns. The returned value can be either GrB_SUCCESS or one of the defined error values shown in Table 2.1. The errors fall into two groups: API errors

(Table 2.1(a)) and execution errors (Table 2.1(b)).

An API error means that a GraphBLAS method was called with parameters that violate the rules for that method. These errors are restricted to those that can be determined by inspecting the types and domains of GraphBLAS objects, GraphBLAS operators, or the values of scalar parameters fixed at the time a method is called. API errors are deterministic and consistent across platforms and implementations. API errors are never deferred, even in nonblocking mode. That is, if a method is called in a manner that would generate an API error, it always returns with the appropriate API error value. If a GraphBLAS method returns with an API error, it is guaranteed that none of the arguments to the method (or any other program data) have been modified.

Execution errors indicate that something went wrong during the execution of a legal GraphBLAS method invocation. Their occurrence may depend on specifics of the executing environment and data values being manipulated. This does not mean that execution errors are the fault of the GraphBLAS implementation. For example, a memory leak could arise from an error in an application's source code (a "program error"), but it may manifest itself in different points of a program's execution (or not at all) depending on the platform, problem size, or what else is running at that time. Index-out-of-bounds and insuficient space execution errors always indicate a program error.

In blocking mode, where each method executes to completion, a returned execution error value applies to the specific method. If a GraphBLAS method, executing in blocking mode, returns with any execution error from Table 2.1(b) other than GrB_PANIC, it is guaranteed that no argument used as input-only has been modified. Output arguments may be left in an invalid state, and their use downstream in the program flow may cause additional errors. If a GraphBLAS method returns with a GrB_PANIC execution error, no guarantees can be made about the state of any program data.

In nonblocking mode, execution errors can be deferred. A return value of GrB_SUCCESS only guarantees that there are no API errors in the method invocation. If an execution error value is returned by a method in nonblocking mode, it indicates that an error was found during execution of the sequence, up to and including the GrB_wait() method (Section **??**) call that ends the sequence. When possible, that return value will provide information concerning the cause of the error.

As discussed in Section **??**, a GrB_wait(obj) on a specific GraphBLAS object obj does not necessarily end a sequence. However, no additional errors on the methods of the sequence that have obj as an OUT or INOUT argument can be reported. From a GraphBLAS perspective, those methods are *complete*.

If a GraphBLAS method, executing in nonblocking mode, returns with any execution error from Table 2.1(b) other than GrB_PANIC, it is guaranteed that no argument used as input-only through the entire sequence has been modified. Any output argument in the sequence may be left in an invalid state and its use downstream in the program flow may cause additional errors. If a GraphBLAS method returns with a GrB_PANIC, no guarantees can be made about the state of any program data.

After a call to any GraphBLAS method, the program can retrieve additional error information (beyond the error code returned by the method) though a call to the function GrB_error(). The signature of that function is shown in Figure 2.1. The function returns a pointer to a NULL-terminated string, and the contents of that string are implementation dependent. In particular, a

```
const char *GrB_error();
```

Figure 2.1: Signature of GrB_error() function.

null string (not a NULL pointer) is always a valid error string. The pointer is valid until the next call to any GraphBLAS method by the same thread. GrB_error() is a thread-safe function, in the sense that multiple threads can call it simultaneously and each will get its own error string back, referring to the last GraphBLAS method it called.

Table 2.1: Error values returned by GraphBLAS methods.

(a) API errors

Error code	Description
GrB_UNINITIALIZED_OBJECT	A GraphBLAS object is passed to a method before new was called on it.
GrB_NULL_POINTER	A NULL is passed for a pointer parameter.
GrB_INVALID_VALUE	Miscellaneous incorrect values.
GrB_INVALID_INDEX	Indices passed are larger than dimensions of the matrix or vector being accessed.
GrB_DOMAIN_MISMATCH	A mismatch between domains of collections and operations when user-defined domains are in use.
GrB_DIMENSION_MISMATCH	Operations on matrices and vectors with incompatible dimensions.
GrB_OUTPUT_NOT_EMPTY	An attempt was made to build a matrix or vector using an output object that already contains valid tuples (elements).
GrB_NO_VALUE	A location in a matrix or vector is being accessed that has no stored value at the specified location.

(b) Execution errors

Error code	Description
GrB_OUT_OF_MEMORY	Not enough memory for operations.
GrB_INSUFFICIENT_SPACE	The array provided is not large enough to hold output.
GrB_INVALID_OBJECT	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	Reference to a vector or matrix element that is outside the defined dimensions of the object.
GrB_PANIC	Unknown internal error.

Chapter 3

Objects

The LAGraph library depends on a number of objects to represent graphs, vectors, and other types associated with graph algorithms in LAGraph. Other objects are not directly associated with the input and output variables, but instead modify the behavior of the LAGraph functions. These are related to the descriptors in the GraphBLAS.

In this chapter, we need to describe all of the objects a user of LAGraph needs to understand. This is also where we describe the types of LAGraph objects and any constraints on those types.

Chapter 4

Functions

The LAGraph library is composed of the following groups of functions:

- Context: Functions that manage the context or environment of an instance of the LAGraph library.
- Graph Algorithms: Functions that implement a Graph Algorithm.
- Utilities: Functions that support implementation of Graph Algorithms or support users of LAGraph.

We need to discuss the rules used in naming the functions and defining their argument lists.

4.1 Context

LAGraph init, finalize and other functions that manage the environment of an instance of LAGraph.

4.2 Graph Algorithms

List the algorithms here. Then have a subsection with the definition of each algorithms.

4.2.1 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 ??. 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 **??**.

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 **??** 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 $\mathsf{mask} \neq \mathsf{GrB\_NULL}$,

      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 ? $\mathsf{A}^T : \mathsf{A}$.

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.

4.3 Utilities

Import, Export, and other functions to support users and LAGraph algorithm developers.

Appendix A

Revision History

Changes in 1.3.1:

- (Issue 70,67) [PENDING] changes to GrB_wait(obj).
- (Issue 69) Made names/symbols containing underscores searchable in PDF.
- Typographical change to eWiseAdd Description to be consistent in order of set intersections.

Appendix B

Examples

B.1 Example: level breadth-first search (BFS) in GraphBLAS

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

/*
 * Given a boolean n x n adjacency matrix A and a source vertex s, performs a BFS traversal
 * of the graph and sets v[i] to the level in which vertex i is visited (v[s] == 1).
 * If i is not reacheable from s, then v[i] = 0. (Vector v should be empty on input.)
 */
GrB_Info BFS(GrB_Vector *v, GrB_Matrix A, GrB_Index s)
{
  GrB_Index n;
  GrB_Matrix_nrows(&n,A);                         // n = # of rows of A

  GrB_Vector_new(v,GrB_INT32,n);                  // Vector<int32_t> v(n)

  GrB_Vector q;                                   // vertices visited in each level
  GrB_Vector_new(&q,GrB_BOOL,n);                  // Vector<bool> q(n)
  GrB_Vector_setElement(q,(bool)true,s);          // q[s] = true, false everywhere else

  /*
   * BFS traversal and label the vertices.
   */
  int32_t d = 0;                                  // d = level in BFS traversal
  bool succ = false;                              // succ == true when some successor found
  do {
    ++d;                                          // next level (start with 1)
    GrB_assign(*v,q,GrB_NULL,d,GrB_ALL,n,GrB_NULL);   // v[q] = d
    GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,
            q,A,GrB_DESC_RC);                     // q[!v] = q ||.&& A ; finds all the
                                                  // unvisited successors from current q
    GrB_reduce(&succ,GrB_NULL,GrB_LOR_MONOID_BOOL,
               q,GrB_NULL);                       // succ = ||(q)
  } while (succ);                                 // if there is no successor in q, we are done.

  GrB_free(&q);                                   // q vector no longer needed

  return GrB_SUCCESS;
}
```

Bibliography

[1] Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[2] Jeremy Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra.* SIAM Press, 2011.