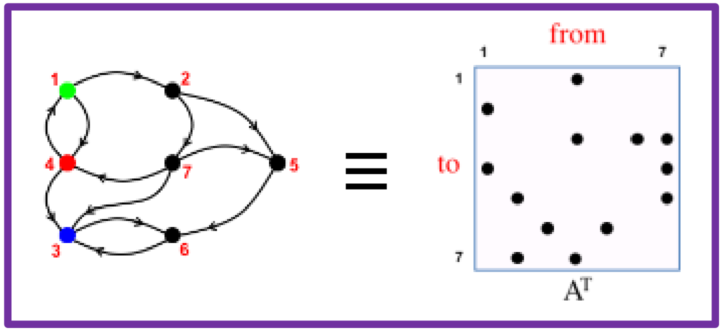
This version includes notes and updates from our first run of this tutorial at HPEC'18



A Hands-On Introduction to the GraphBLAS

http://graphblas.org

Scott McMillan

CMU/SEI

Tim Mattson

Intel Labs

Brought to you by the "GraphBLAS C Specification Gang" (of Five):

Aydın Buluç (LBNL), Tim Mattson (Intel), Scott McMillan (CMU/SEI)
Jose Moreira (IBM), Carl Yang (UC Davis)

… and a special thank you to **Tim Davis (Texas A&M)** for GraphBLAS support in SuiteSparse

DM18-1037

HPEC tutorial

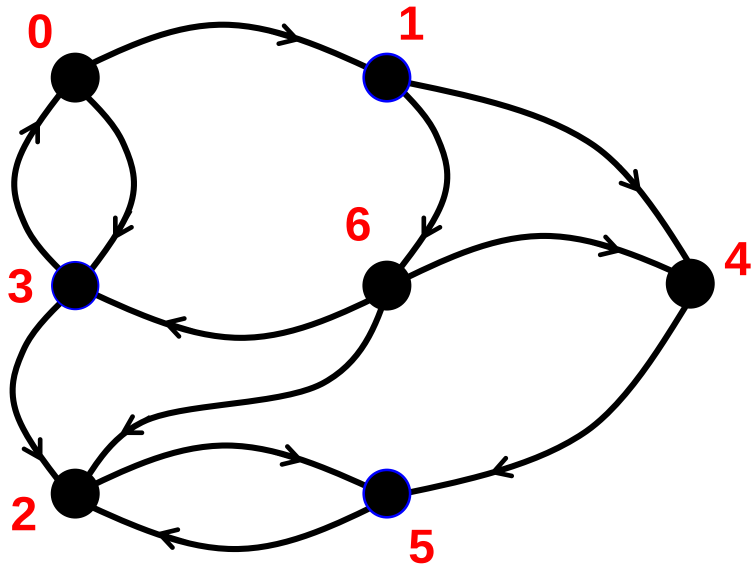
- We had 16 students by the end of the day.
- Two of them had Windows systems and were not able to fully participate. They stayed through the whole tutorial (I guess they did the exercises but didn’t compiler or run them).
- One person (not included in that count of 16) at the beginning of the tutorial when he heard we do not support windows, left to attend a different tutorial.
- We took about two hours to get up to the section on Breadth first traversal which is pretty much exactly what we expected. We did skip exercise 6 which was just fine.
- We made it to the end of the tutorial.   Some students actually finished the BFS levels exercise.

Outline

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices
- GraphBLAS Operations
- Breadth-First Traversal

Understanding relationships between items

- Graph: A visual representation of a set of vertices and the connections between them (edges).



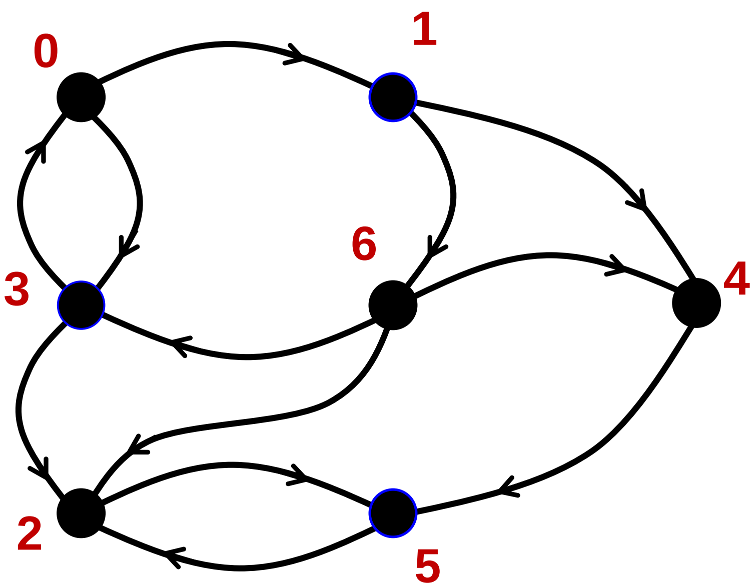
- Graph: Two sets, one for the vertices (v) and one for the edges (e)

$$v \in [0, 1, 2, 3, 4, 5, 6]$$

$$e \in [(0,1), (0,3), (1,4), (1,6), (2,5), (3,0), (3,2), (4,5), (5,2), (6,2), (6,3), (6,4)]$$

A graph as a matrix

- Adjacency Matrix: A square matrix (usually sparse) where rows and columns are labeled by vertices and non-empty values are edges from a row vertex to a column vertex



To vertex (columns)

$$A = \begin{pmatrix} - & \star & - & \star & - & - & - \\ - & - & - & - & \star & - & \star \\ - & - & - & - & - & \star & - \\ \star & - & \star & - & - & - & - \\ - & - & - & - & - & \star & - \\ - & - & \star & - & - & - & - \\ - & - & \star & \star & \star & - & - \end{pmatrix}$$

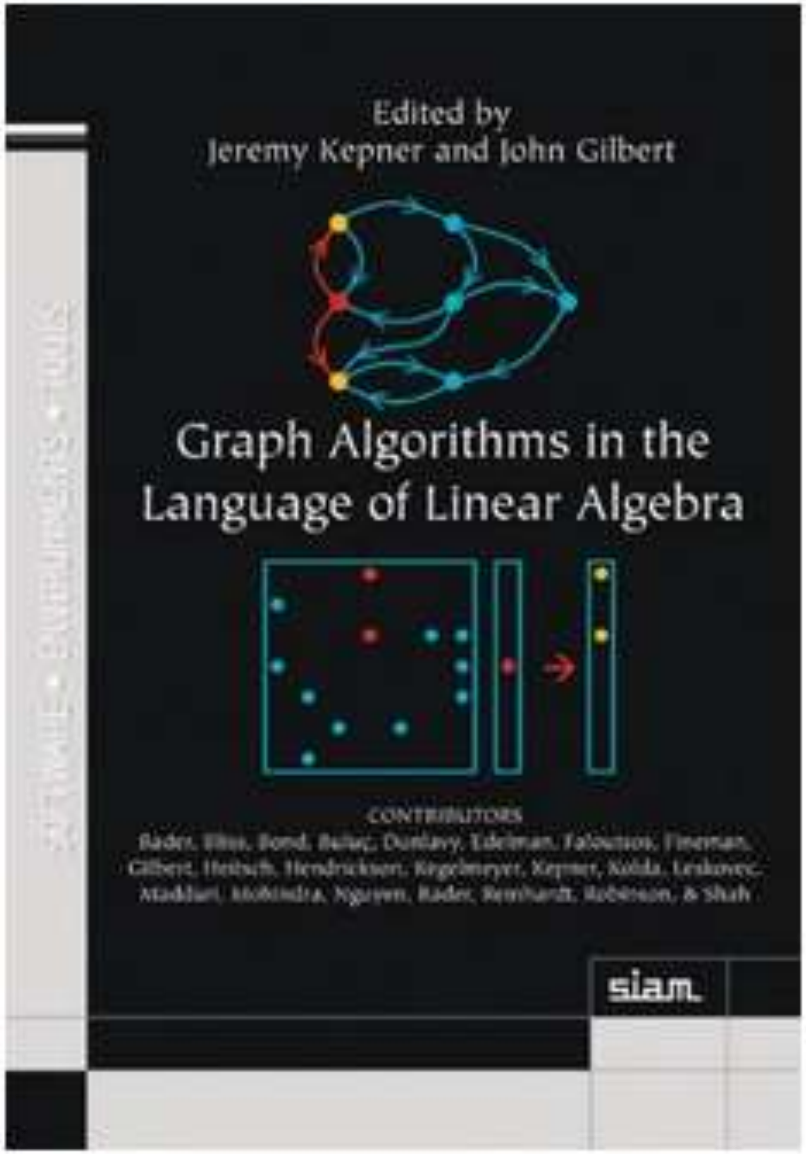
From vertex (rows)

By using a matrix, I can turn algorithms working with graphs into linear algebra.

Graph Algorithms and Linear Algebra

- Most common graph algorithms can be represented in terms of linear algebra.
  - This is a mature field … it even has a book.

- Benefits of graphs as linear algebra
  - Well suited to memory hierarchies of modern microprocessors
  - Can utilize decades of experience in distributed/parallel computing from linear algebra in supercomputing.
  - Easier to understand … for some people.



How do linear algebra people write software?

- They do so in terms of the BLAS:
  - The **B**asic **L**inear **A**lgebra **S**ubprograms: low-level building blocks from which any linear algebra algorithm can be written

BLAS 1	Vector/vector	Lawson, Hanson, Kincaid and Krogh, 1979	LINPACK
BLAS 2	Matrix/vector	Dongarra, Du Croz, Hammarling and Hanson, 1988	LINPACK on vector machines
BLAS 3	Matrix/matrix	Dongarra, Du Croz, Hammarling and Hanson, 1990	LAPACK on cache based machines

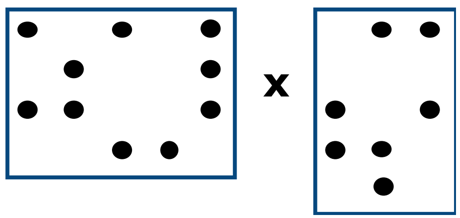
- The BLAS supports a separation of concerns:
  - HW/SW optimization experts tuned the BLAS for specific platforms.
  - Linear algebra experts built software on top of the BLAS ... high performance “for free”.

- It is difficult to over-state the impact of the BLAS … they revolutionized the practice of computational linear algebra.

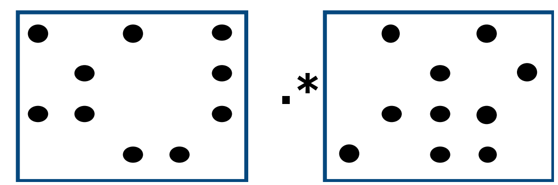
GraphBLAS: building blocks for graphs as linear algebra

- Basic objects
  - Matrix, vector, algebraic structures, and ”control objects”
- Fundamental operations over these objects

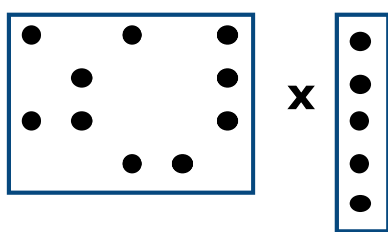
Matrix multiplication



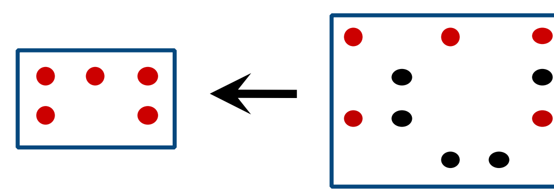
Element-wise operations (eWiseAdd, eWiseMult)



Matrix-vector multiplication (vxm, mxv)



Extract (and Assign) submatrices



…plus reductions, transpose, and application of a function to each element of a matrix or vector

GraphBLAS References

Mathematical Foundations of the GraphBLAS

Jeremy Kepner (MIT Lincoln Laboratory Supercomputing Center), Peter Aaltonen (Indiana University), David Bader (Georgia Institute of Technology), Aydın Buluç (Lawrence Berkeley National Laboratory), Franz Franchetti (Carnegie Mellon University), John Gilbert (University of California, Santa Barbara), Dylan Hutchison (University of Washington), Manoj Kumar (IBM), Andrew Lumsdaine (Indiana University), Henning Meyerhenke (Karlsruhe Institute of Technology), Scott McMillan (CMU Software Engineering Institute), Jose Moreira (IBM), John D. Owens (University of California, Davis), Carl Yang (University of California, Davis), Marcin Zalewski (Indiana University), Timothy Mattson (Intel)

IEEE HPEC 2016

Design of the GraphBLAS API for C

Aydın Buluç[†], Tim Mattson[‡], Scott McMillan[§], José Moreira[¶], Carl Yang[*,†]

[†] *Computational Research Division, Lawrence Berkeley National Laboratory*
[‡] *Intel Corporation*
[§] *Software Engineering Institute, Carnegie Mellon University*
[¶] *IBM Corporation*
[*] *Electrical and Computer Engineering Department, University of California, Davis, USA*

IEEE HPEC 2017

The official GraphBLAS C spec can be found at: www.graphblas.org

GraphBLAS Implementations

- Multiple implementation projects:
  - SuiteSparse library, http://faculty.cse.tamu.edu/davis/suitesparse.html
  - A C-wrapper around the GPU Gunrock library from UC Davis, http://adsabs.harvard.edu/abs/2017arXiv170101170W
  - The IBM GraphBLAS C implementation, https://github.com/IBM/ibmgraphblas
  - CMU/SEI C++ GraphBLAS Template Library https://github.com/cmu-sei/gbtl (there is also a python wrapper called pyGB with UW/PNNL).
- We'll use SuiteSparse:GraphBLAS in this tutorial

SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra, T. A. Davis, Texas A&M U, submitted to TOMS

Exercise 1: Build a GraphBLAS program

- Clone our git repository
- Includes the following components
  - Exercises and solutions
  - SuiteSparse library, binaries for Linux and OSX and source
- Load software onto your system, make sure you can build and run our test program

```
$ git clone https://github.com/tgmattso/GraphBLAS.git
$ cd GraphBLAS/src
$ make BuildGraph.exe
$ ./BuildGraph.exe
```

- If all goes well, your output should look like this:

```
$ ./BuildGraph.exe
Matrix: GRAPH =
[  -,   -,   -]
[  -,   -,   4]
[  -,   -,   -]
```

Outline

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices
- GraphBLAS Operations
- Breadth-First Traversal

GraphBLAS C API

- A binding of the GraphBLAS math to the C programming language.

- Requires C99 extended with function polymorphism based on static-types and number-of-parameters.
  - All modern C compilers in common use today support these extensions

- Basic include file with function prototypes, types, and constants
  - `#include <GraphBLAS.h>`

- Includes a few types and opaque objects (e.g. matrices and vectors) to give implementations maximum flexibility

  `GrB_Index` → An integer type used to set dimensions and index into arrays

  `GrB_Matrix` → A 2D sparse array, row indices, column indices and values

  `GrB_Vector` → A 1D sparse array

  - … plus additional opaque objects we'll describe later (descriptors, semirings, binary operators, and unary operators)

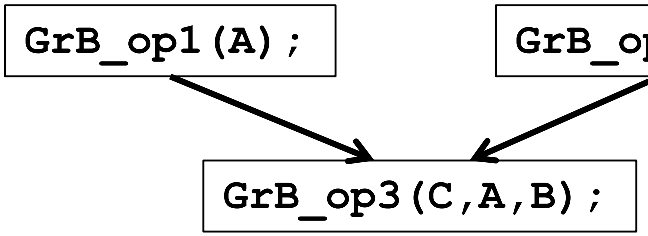
GraphBLAS C API: Basic definitions

- **Opaque object**: An object manipulated strictly through the GraphBLAS API whose implementation is not defined by the GraphBLAS specification.
- **Transparent object**: an object whose structure is fully exposed to the programmer. E.g.: an array of tuples <i, j, value>
- **Method**: Any C function that manipulates a GraphBLAS opaque object.
- **Domain**: the set of available values used for the elements of matrices, the elements of vectors, and when defining operators.
  - Examples are **`GrB_UINT64, GrB_INT32, GrB_BOOL, GrB_FP32`**
- **Operation**: a method that corresponds to an operation defined in the GraphBLAS math spec. http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf
  - Examples: matrix multiply, matrix-vector multiply, reduction, apply

Execution modes

- A GraphBLAS program defines a DAG of operations.
- Objects are defined by the sequence of GraphBLAS method calls, but the value of the object is not assured until a GraphBLAS method queries its state.
- This gives an implementation flexibility to optimize the execution (fusing methods, replacing method sequences by more efficient ones, etc.)

```
GrB_op1(A);
GrB_op2(B);
GrB_op3(C,A,B);
```



- An execution of a GraphBLAS program defines a context for the library.
- The execution runs in one of two modes:
  - Blocking mode … executes methods in program order with each method completing before the next is called
  - Non-Blocking mode … methods launched in order. Complete in any order consistent with the DAG. Objects do not exit in fully defined state until queried.
- Most implementations only support blocking mode.

Predefined low-level types

- Predefined types used to define domains in GraphBLAS

GrB_Type values	C type	domain
GrB_BOOL	bool	$\{\mathtt{false}, \mathtt{true}\}$
GrB_INT8	int8_t	$\mathbb{Z} \cap [-2^7, 2^7)$
GrB_UINT8	uint8_t	$\mathbb{Z} \cap [0, 2^8)$
GrB_INT16	int16_t	$\mathbb{Z} \cap [-2^{15}, 2^{15})$
GrB_UINT16	uint16_t	$\mathbb{Z} \cap [0, 2^{16})$
GrB_INT32	int32_t	$\mathbb{Z} \cap [-2^{31}, 2^{31})$
GrB_UINT32	uint32_t	$\mathbb{Z} \cap [0, 2^{32})$
GrB_INT64	int64_t	$\mathbb{Z} \cap [-2^{63}, 2^{63})$
GrB_UINT64	uint64_t	$\mathbb{Z} \cap [0, 2^{64})$
GrB_FP32	float	IEEE 754 binary32
GrB_FP64	double	IEEE 754 binary64

Code from our first example

```
#include <stdio.h>
#include <assert.h>
#include <GraphBLAS.h>
#include "tutorial_utils.h"

int main(int argc, char** argv) {
   GrB_init(GrB_BLOCKING);
   GrB_Index const NUM_NODES = 3;
   GrB_Matrix graph;
   GrB_Matrix_new(&graph, GrB_UINT64,
                  NUM_NODES, NUM_NODES);
   GrB_Matrix_setElement(graph, 4, 1, 2);

   pretty_print_matrix_UINT64(graph, "GRAPH");
   GrB_Index nvals;
   GrB_Matrix_nvals(&nvals, graph);
   assert(nvals == 1);

   // Cleanup
   GrB_free(&graph);
   GrB_finalize();
}
```

Initialize a context in BLOCKING mode

GrB_Index used for matrix dimension

Create a matrix object of order **NUM_NODES** and domain **UINT64**

Store the value 4 in element (1,2)

Our own "pretty print" routine (not part of GraphBLAS)

Query the matrix for the number of defined (stored) values and check for correctness

Free memory used for our matrix

Close the context, release resources

Exercise 2: Adjacency matrix

- Draw a simple graph with 3 to 5 nodes.
- Write a program to create the adjacency matrix.
  - Use BuildGraph.c as an example.
- Output the result and verify that your adjacency graph is correct.
- You will need the following types and methods from the GraphBLAS
  - **`GrB_Index,   GrB_Matrix`**
  - **`GrB_init();  GrB_finalize();`**
  - **`GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);`**
  - **`GrB_Matrix_setElement(graph, value, from_node, to_node);`**
  - **`GrB_Matrix_nvals(&nvals, graph);`**
  - **`GrB_free(&graph);`**

- Hint: Save time and minimize typing
  - Copy BuildGraph.c into another file (e.g. exercise2.c) and modify it to build your adjacency matrix program.
  - Edit the makefile and add your new source file to the list in the definition of SOURCES. Then you can just type “make” to build your program.

Exercise 2: Adjacency matrix

- Draw a simple graph with 3 to 5 nodes.
- Write a program to create the adjacency matrix.
  - Use BuildGraph.c as an example.
- Output the result and verify that your adjacency graph is correct.
- You will need the following types and methods from the GraphBLAS
  - **`GrB_Index,   GrB_Matrix`**
  - **`GrB_init();  GrB_finalize();`**
  - **`GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);`**
  - **`GrB_Matrix_setElement(graph, value, from_node, to_node);`**
  - **`GrB_Matrix_nvals(&nvals, graph);`**
  - **`GrB_free(&graph);`**
- Hint: Save time and minimize typing
  - Copy BuildGraph.c into another file (e.g. exercise1.c) and modify it to build your adjacency matrix program.
  - Edit the makefile and add your new source file to the list in the definition of SOURCES. Then you can just type “make” to build your program.

A quick API note … Opaque objects are passed around through a handle (e.g. graph).

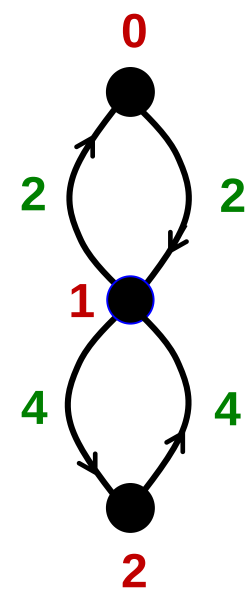
When the handle itself changes, we pass by address (i.e. with a &).

When the object referenced by the handle is manipulated but the handle doesn’t change, we pass by value (i.e. without the &).

Solution to Exercise 2

```
...
GrB_init(GrB_BLOCKING);

GrB_Index const NUM_NODES = 3;
GrB_Matrix graph;
GrB_Matrix_new(&graph, GrB_UINT64,
               NUM_NODES, NUM_NODES);

GrB_Matrix_setElement(graph, 4, 1, 2);
GrB_Matrix_setElement(graph, 4, 2, 1);
GrB_Matrix_setElement(graph, 2, 0, 1);
GrB_Matrix_setElement(graph, 2, 1, 0);

pretty_print_matrix_UINT64(graph, "Graph");

GrB_free(&graph);
GrB_finalize();
```

Our three node graph with edge weights:




```
Matrix: Graph:
[ -,  2,  -]
[ 2,  -,  4]
[ -,  4,  -]
```

Building matrices

- Building a matrix one edge at a time is awkward.
- It is often more convenient to do it from vectors defining the indices and values for non-empty elements of the sparse matrix

```
GrB_Info GrB_Matrix_build( GrB_Matrix          C,
                           const GrB_Index     *row_indices,
                           const GrB_Index     *col_indices,
                           const <type>        *values,
                           GrB_Index            n,
                           const GrB_BinaryOp   dup);
```

- **row_indices**, **col_indices**, and **values** are transparent arrays.
- **<type>** is a C type consistent with the domain of the matrix
- **n** is the number of entries in the sparse matrix
- **dup** is an associative, commutative function to apply to the values should duplicate locations be specified.
  - Typically use one of the GraphBLAS predefined operators

Building matrices

- Building a matrix one edge at a time is awkward.
- It is often more convenient to do it from vectors defining the indices and values for non-empty elements of the sparse matrix

```
GrB_Info GrB_Matrix_build( GrB_Matrix          C,
                           const GrB_Index     *row_indices,
                           const GrB_Index     *col_indices,
                           const <type>        *values,
                           GrB_Index            n,
                           const GrB_BinaryOp   dup);
```

Return values:
- GrB_SUCCESS if everything worked
- Other values for problems with input arguments, memory issues, internal errors or other problems.

- **`row_indices`**, **`col_indices`**, and **`values`** are transparent arrays.
- **`<type>`** is a C type consistent with the domain of the matrix
- **`n`** is the number of entries in the sparse matrix
- **`dup`** is an associative, commutative function to apply to the values should duplicate locations be specified.
  - Typically use one of the GraphBLAS predefined operators

GraphBLAS predefined operators

- A subset of operators from Table 2.3 of the GraphBLAS specification

Identifier	Domains	Description	
GrB_LOR	bool x bool → bool	$f(x,y) = x \vee y$	Logical OR
GrB_LAND	bool x bool → bool	$f(x,y) = x \wedge y$	Logical AND
GrB_EQ_*T*	*T* x *T* → bool	$f(x,y) = (x==y)$	Equal
GrB_MIN_*T*	*T* x *T* → *T*	$f(x,y) =(x<y)?x:y$	minimum
GrB_MAX_*T*	*T* x *T* → *T*	$f(x,y) =(x>y)?x:y$	maximum
GrB_PLUS_*T*	*T* x *T* → *T*	$f(x,y) = x + y$	addition
GrB_TIMES_*T*	*T* x *T* → *T*	$f(x,y) = x * y$	multiplication
GrB_FIRST_*T*	*T* x *T* → *T*	$f(x,y) = x$	First argument
GrB_SECOND_*T*	*T* x *T* → *T*	$f(x,y) = y$	Second argument

Where *T* is a suffix indicating type and includes **`FP32`**, **`FP64`**, **`INT32`**, **`UINT32`**, **`BOOL`**
Note: **`GrB_FIRST`** and **`GrB_SECOND`** are not commutative operators

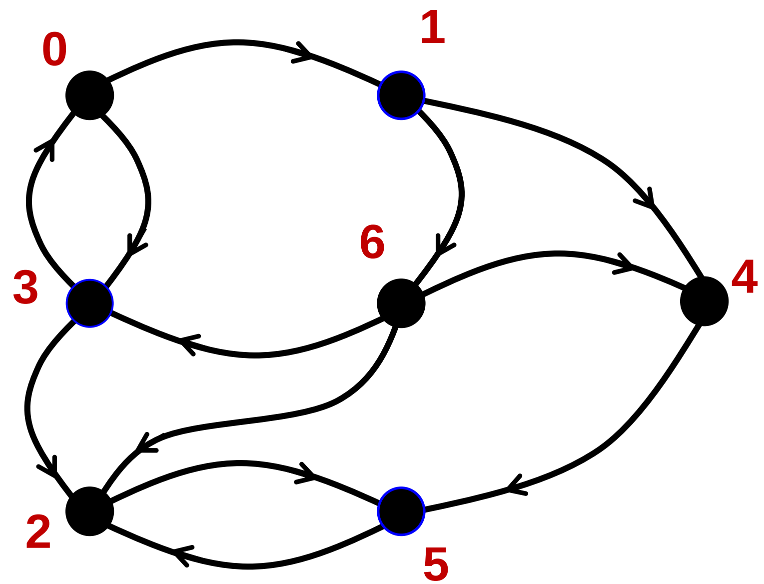
This is a subset of the defined types and operators. See table 2.3 for the full list.

C code fragment using GrB_Matrix_build

```
GrB_Index const NUM_NODES = 3;
GrB_Index const NUM_EDGES = 4;

GrB_Index row_indices[] = {0, 1, 1, 2};
GrB_Index col_indices[] = {1, 0, 2, 1};
bool values[] = {true, true, true, true};
GrB_Matrix graph;

GrB_Matrix_new(&graph, GrB_BOOL, NUM_NODES, NUM_NODES);

GrB_Matrix_build(graph,
                 row_indices, col_indices, (bool*)values,
                 NUM_EDGES, GrB_LOR);
```

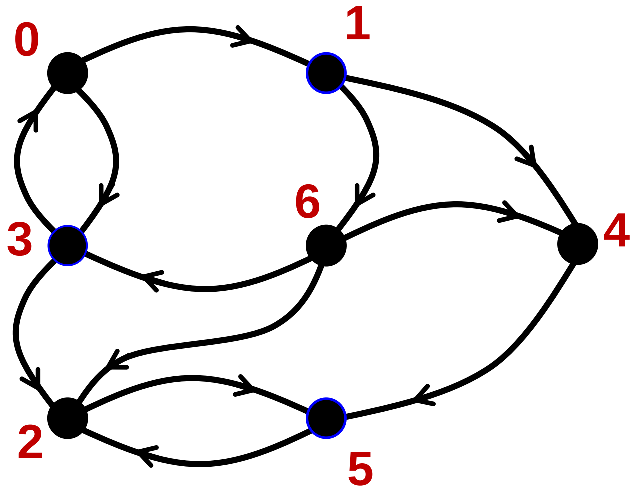
Exercise 3: Adjacency matrix

- Write a program to create the adjacency matrix for the GraphBLAS "logo" graph using row, column and value arrays.



- You will need the following types and methods from the GraphBLAS
  - **`GrB_Index,   GrB_Matrix`**
  - **`GrB_init();  GrB_finalize();`**
  - **`GrB_Matrix_new(&graph, GrB_domain, Nrows, Ncols);`**
  - **`GrB_Matrix_build(graph, row_indices, col_indices, values, NUM_EDGES, dup);`**
  - **`GrB_Matrix_nvals(&nvals, graph);`**
  - **`GrB_free(&graph);`**

Summary of solution to exercise 3




```
Matrix: Graph =
[ -,  1,  -,  1,  -,  -,  -]
[ -,  -,  -,  -,  1,  -,  1]
[ -,  -,  -,  -,  -,  1,  -]
[ 1,  -,  1,  -,  -,  -,  -]
[ -,  -,  -,  -,  -,  1,  -]
[ -,  -,  1,  -,  -,  -,  -]
[ -,  -,  1,  1,  1,  -,  -]
```

```
GrB_Index const NUM_NODES = 7;
GrB_Index const NUM_EDGES = 12;

GrB_Index row_indices[] = {0, 0, 1, 1, 2, 3, 3, 4, 5, 6, 6, 6};
GrB_Index col_indices[] = {1, 3, 4, 6, 5, 0, 2, 5, 2, 2, 3, 4};
bool values[] = {true, true, true, true, true, true,
                 true, true, true, true, true, true};
GrB_Matrix graph;
GrB_Matrix_new(&graph, GrB_BOOL, NUM_NODES, NUM_NODES);
GrB_Matrix_build(graph, row_indices, col_indices, (bool*)values,
                 NUM_EDGES, GrB_LOR);
pretty_print_matrix_UINT64(graph, "Graph");
```

Outline

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices
- GraphBLAS Operations
- Breadth-First Traversal

GraphBLAS Operations (from the Math Spec*)

Operation name	Mathematical description
mxm	$\mathbf{C} \odot= \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w} \odot= \mathbf{A} \oplus . \otimes \mathbf{v}$
vxm	$\mathbf{w}^T \odot= \mathbf{v}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C} \odot= \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C} \odot= \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w} \odot= \bigoplus_j \mathbf{A}(:, j)$
apply	$\mathbf{C} \odot= F_u(\mathbf{A})$
	$\mathbf{w} \odot= F_u(\mathbf{u})$
transpose	$\mathbf{C} \odot= \mathbf{A}^T$
extract	$\mathbf{C} \odot= \mathbf{A}(\mathbf{i}, \mathbf{j})$
	$\mathbf{w} \odot= \mathbf{u}(\mathbf{i})$
assign	$\mathbf{C}(\mathbf{i}, \mathbf{j}) \odot= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot= \mathbf{u}$

We use $\odot$, $\oplus$, and $\otimes$ since later on we'll manipulate the algebraic structure to generalize them to other operations.

* Mathematical foundations of the GraphBLAS, Kepner et. al. HPEC'2016

GrB_mxv()

$$w \odot= A \oplus . \otimes u$$

Multiply a matrix times a vector to produce a vector

$$w(i) = w(i) \odot \sum_{k=0}^{N} A(i,k) \otimes u(k)$$

$$w \in S^M \qquad u \in S^N \qquad A \in S^{M \times N}$$

Definitions:

- S is the domain of the objects w, u, and A
- $\odot$ is an optional accumulation operator (a binary operator)
- $\otimes$ and $\oplus$ are multiplication and addition (or generalizations thereof)
- Σ uses the $\oplus$ operator

GrB_mxv()

$$w \odot= A \oplus . \otimes u$$

Multiply a matrix times a vector to produce a vector

$$w(i) = w(i) \odot \sum_{\boldsymbol{k \in ind(A(i,:)) \cap ind(u)}} A(i,k) \otimes u(k)$$

The summation is over the intersection of the existing elements in the i^{th} row of A with u … which avoids exposing how empty elements (i.e. "zeros") are represented. This becomes important when we change the semiring between operations

$$w \in S^M \qquad u \in S^N \qquad A \in S^{M \times N}$$

Definitions:

- S is the domain of the objects w, u, and A
- $\odot$ is an optional accumulation operator (a binary operator)
- $\otimes$ and $\oplus$ are multiplication and addition (or generalizations thereof)
- Σ uses the $\oplus$ operator
- ind(u) returns the indices of the stored values of u

GrB_mxv()

$$w \odot= A \oplus . \otimes u$$

- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

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

GrB_mxv()

$$w \cancel{\otimes}= A \oplus . \otimes u$$

- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

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

Let's ignore mask, accum and desc for now and use default values (indicated by GrB_NULL)

GrB_mxv()

$$w \cancel{\otimes}= A \oplus . \otimes u$$

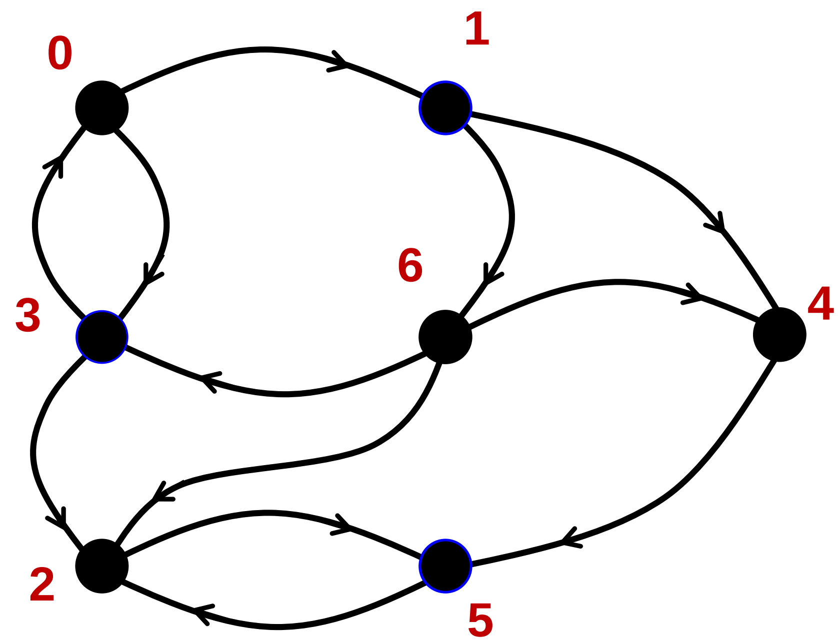
- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

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

Op defines the algebraic structure, a semiring in this case. This gives us ⊗ and ⊕ and the identity for ⊕. We'll say much more about his later.
For our first exercises with bool objects, we'll use a built-in SuiteSparse semiring GxB_LOR_LAND_BOOL.

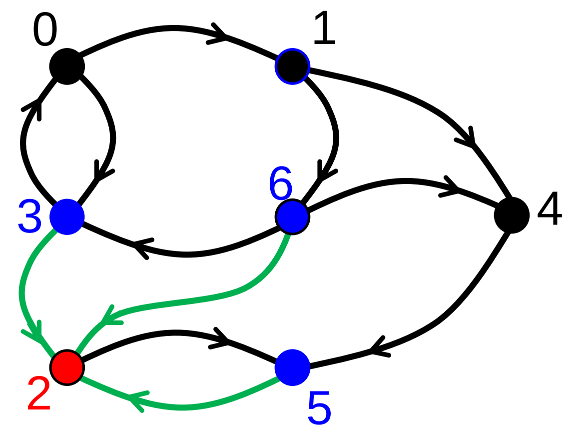
Exercise 4: Matrix Vector Multiplication

- Use the adjacency matrix from exercise 3 and a vector with a single value to select one of the nodes in the graph.
- Find the product mxv, print the result, and interpret its meaning.
- In addition to those from Exercise 3, you'll need the functions:



  - `GrB_Vector result, vec;`
  - `GrB_Index NODE;`
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - `pretty_print_vector_UINT64(vec, ”Input node");`
  - `GrB_mxv(result, GrB_NULL, GrB_NULL,`
    `GxB_LOR_LAND_BOOL, graph, vec, GrB_NULL);`

Solution to exercise 4




```
pretty_print_matrix_UINT64(graph, "GRAPH");

// Build a vector with one node set.
GrB_Index const NODE = 2;
GrB_Vector vec, result;
GrB_Vector_new(&result, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(vec, true, NODE);
pretty_print_vector_UINT64(vec, "Target node");

GrB_mxv(result, GrB_NULL, GrB_NULL,
        GxB_LOR_LAND_BOOL, graph, vec, GrB_NULL);
pretty_print_vector_UINT64(result, "sources");
```

```
Matrix: GRAPH =
[ -,  1,  -,  1,  -,  -,  -]
[ -,  -,  -,  -,  1,  -,  1]
[ -,  -,  -,  -,  -,  1,  -]
[ 1,  -,  1,  -,  -,  -,  -]
[ -,  -,  -,  -,  -,  1,  -]
[ -,  -,  1,  -,  -,  -,  -]
[ -,  -,  1,  1,  1,  -,  -]
Vector: Target node =
[ -,  -,  1,  -,  -,  -,  -]
Vector: sources =
[ -,  -,  -,  1,  -,  1,  1]
```

The stored elements of the adjacency matrix, a(i,j) indicate an edge from vertex i to vertex j

So the matrix vector product scans over a row (from) to find when an edge lands at the destination

Finding neighbors

- A more common operation is to input a vector selecting a source and find all the neighbors one hop away from that vertex.
- Using mxv(), how would you do this?

Finding neighbors

- A more common operation is to input a vector selecting a source and find all the neighbors one hop away from that vertex.
- Using mxv(), how would you do this?
  - The adjacency matrix elements indicate edges
    - from a vertex (row index)
    - to another vertex (columns index)
  - Then the transpose of the adjacency matrix indicates edges
    - To a vertex (row index)
    - From other vertices (column index)
- Therefore, we can find the neighbors of a vertex (marked by the non-empty elements of v)

$$\text{Neighbors} = A^T \oplus . \otimes v$$

- The GraphBLAS defines a transpose operation, but given how often you need to do a transpose, there must be a better way

Changing the behavior of a GraphBLAS operation

- Most GraphBLAS operations take an argument that is an opaque object called a "descriptor". You declare an descriptor called "desc" and create it as follows:

```
GrB_Descriptor desc;
GrB_Descriptor_new (&desc);
```

- The descriptor controls the behavior of the method and how objects are handled inside the method.
- The descriptor controls:
  - Do you *transpose input matrices*? (GrB_TRAN)
  - Does the computation *replace existing values in the output object* or combine with them? (GrB_REPLACE)
  - Take the *structural complement of the mask* object (swap empty/false ←→ filled/true values in a sparse object). (GrB_SCMP)

....To be discussed later

Using Descriptors

- A descriptor is an opaque object so you set its values with a GraphBLAS method.
- A descriptor *field* selects the object it impacts:
  - **`GrB_OUTP`**: The output GraphBLAS object
  - **`GrB_INP0`**: The first input GraphBLAS object (matrix or vector)
  - **`GrB_INP1`**: The second input GraphBLAS object (matrix or vector)
  - **`GrB_MASK`**: The GraphBLAS mask object (described later).
- A descriptor *value* describes the action to be taken.

- For example, to transpose the first input matrix, you'd call the operation and pass in the following descriptor

```
GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);
```

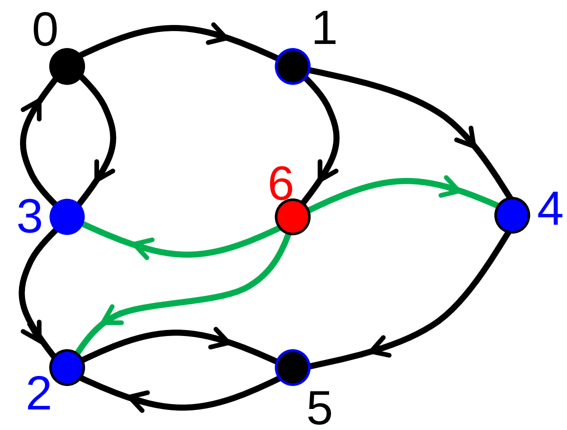
Exercise 5: Matrix Vector Multiplication

- Modify your program from exercise 4 to multiply by the transpose of the adjacency matrix.
- Verify that you can use that to find the one-hop neighbors of any vertex

```
– GrB_Vector result, vec;
– GrB_Index NODE;
– GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
– GrB_Vector_setElement(vec, true, NODE);
– pretty_print_vector_UINT64(vec, ”Input node");
– GrB_Descriptor desc;
– GrB_Descriptor_new(&desc);
– GrB_Descriptor_set(desc, FIELD, VALUE)
– GrB_mxv(result, GrB_NULL, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, vec, desc);
```

```
FIELD: GrB_INP0, GrB_INP1, GrB_OUTP, GrB_MASK
VALUE: GrB_TRAN, GrB_REPLACE, GrB_SCMP
```

Solution to exercise 5




```
// Build a vector with one node set.
GrB_Index const SRC_NODE = 6;
GrB_Vector vec;
GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(vec, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(vec, "source node");
GrB_mxv(vec, GrB_NULL, GrB_NULL,
        GxB_LOR_LAND_BOOL, graph, vec, desc);
pretty_print_vector_UINT64(vec, "neighbors");
```

```
Vector: source node =
[ -,  -,  -,  -,  -,  -,  1]
Vector: neighbors =
[ -,  -,  1,  1,  1,  -,  -]
GrB_mxv test passed.
```

The transposed matrix vector product scans over a columns (to) to find edges that start at the source node.

GrB_mxv()

$$w \cancel{\otimes}= A \oplus . \otimes u$$

- Compute the product of a GraphBLAS Sparse Matrix with a GraphBLAS vector.
- Returns error codes of type GrB_info. See the spec for details.

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

It's time to explain semirings in GraphBLAS operation

Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative,   Op2 distributes over Op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over Op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

(R, +, *, 0, 1) Real Field	Standard operations in linear algebra

Notation: (R, +, *, 0, 1)

Scalar type, Op1, Op2, Identity Op1, Identity Op2

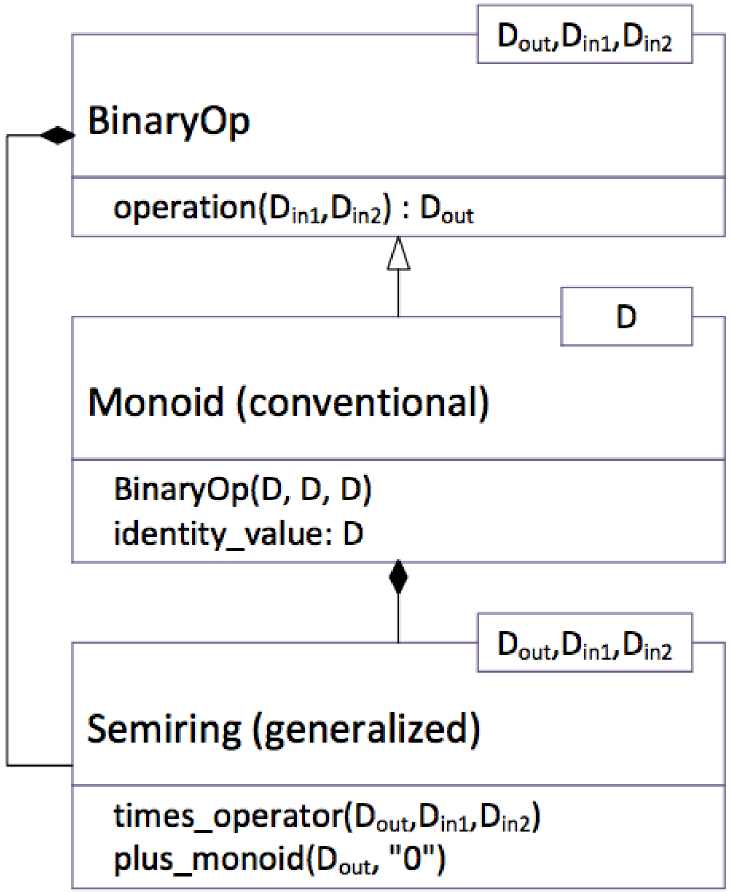
Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing (+,*) with binary operations (Op1, Op2)
  - Op1 and Op2 have identity elements sometimes called 0 and 1
  - Op1 and Op2 are associative.
  - Op1 is commutative, Op2 distributes over Op1 from both left and right
  - The Op1 identify is an Op2 annihilator.

(R, +, *, 0, 1) Real Field	Standard operations in linear algebra
(R U {∞},min, +, ∞, 0) Tropical semiring	Shortest path algorithms
({0,1}, |, &, 0, 1) Boolean Semiring	Graph traversal algorithms
(R U {∞}, min, *, ∞, 1)	Selecting a subgraph or contracting nodes to form a quotient graph.

Algebraic structures in the GraphBLAS:
Semirings and Monoids

- The GraphBLAS semiring defines:
  - A set of allowed values (the domain)
  - Two commutative operators called addition and multiplication
  - An additive identity (called 0) that is the annihilator over multiplication.
- A Monoid is used in defining a semiring:
  - Monoid: A domain, an associative binary operator and an identity corresponding to that operator



Hierarchy of algebraic object classes showing relationships between the various domains and the operators.

Building Semirings in the GraphBLAS

- First you build the monoid (M) for a particular domain, D, the “addition” operator, and its identity:

$$M = < D, \bigoplus, 0 >$$

- Then define the semiring (S) in terms of the Monoid and the multiplications operator:

$$S = < D_{out}, D_{in1}, D_{in2}, M, \otimes >$$

- The domains must be consistent:

$$\otimes\text{:}\ D_{in1} \times D_{in2} \rightarrow D_{out}$$

$$\bigoplus\text{:}\ D_{out} \times D_{out} \rightarrow D_{out}$$

$$0 \in D_{out}$$

Building Semirings in the GraphBLAS

- First you build the monoid (M) for the “addition” and its identity:

```
GrB_Info GrB_Monoid_new( GrB_Monoid          *monoid,
                         GrB_BinaryOp         binary_op,
                         <type>               identity);
```

- Where the type must be consistent with that of the binary operator which is either a built-in operator (Spec. Table 2.3) or a user-defined operator (not covered here)
- Example:

```
GrB_Monoid UInt64Plus ;
GrB_Monoid_new(&UInt64Plus, GrB_PLUS_UINT64, 0 ul);
```

Building Semirings in the GraphBLAS

- Then you build the semiring pairing a monoid (“add”) with a binary operator (“mul”) :

```
GrB_Info GrB_Semiring_new( GrB_Semiring     *semiring,
                           GrB_Monoid        add_op,
                           GrB_BinaryOp      mul_op);
```

- The monoid’s identity *should* be the binary operator’s annihilator (not enforced).

- Example using the monoid from the previous page:

```
GrB_Semiring UInt64Arith;
GrB_Semiring_new(&UInt64Arith, UInt64Plus, GrB_TIMES_UINT64);
```

Common Semirings

semiring	Domain	Add	Add-identity	multiply
Boolean	`GrB_BOOL`	`GrB_LOR`	`false`	`GrB_LAND`
Int32 arithmetic	`GrB_INT32`	`GrB_PLUS_INT32`	`0`	`GrB_TIMES_INT32`
FP32 arithmetic	`GrB_FP32`	`GrB_PLUS_FP32`	`0.0f`	`GrB_TIMES_FP32`
Max_second	`GrB_FP32`	`GrB_MAX_FP32`	`0.0f`	`GrB_SECOND_FP32`

Exercise 6: Changing semirings

- Up to this point, we've used a built-in Boolean semiring that is included with SuiteSparse (**`GxB_LOR_LAND_BOOL`**).
- Pick any of the past exercises and experiment with different semi-rings.
  - **`GrB_Monoid UInt64Plus;`**
  - **`GrB_Monoid_new(&UInt64Plus, GrB_PLUS_UINT64, 0ul);`**
  - **`GrB_Semiring UInt64Arith;`**
  - **`GrB_Semiring_new(&UInt64Arith, UInt64Plus, GrB_TIMES UINT64);`**

Outline

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices
- GraphBLAS Operations
- Breadth-First Traversal

Breadth First Traversal

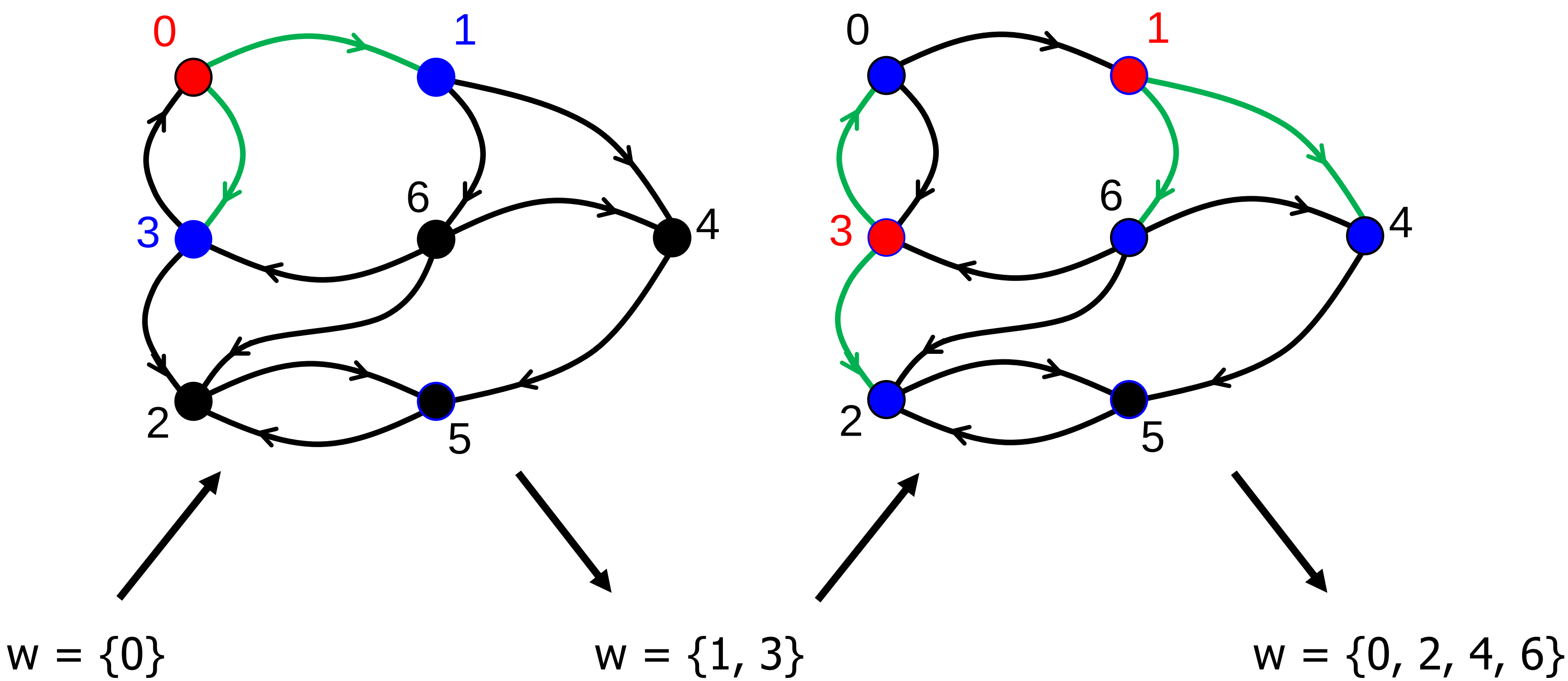
- The Breadth First Traversal:
  - Start from one or more initial vertices
  - Visit all accessible one hop neighbors,
  - Visit all accessible unique two hop neighbors,
  - Continue until no more unique vertices to visit
  - Note: keep track of vertices visited so you don't visit the same vertex more than once
- Breadth first traversal is a common pattern used in a range of graph algorithms
  - Build a spanning tree that contains all vertices and minimal number of edges
  - Search for accessible vertices with certain properties.
  - Find shortest paths between vertices.
  - Other more advanced algorithms such as maxflow and betweenness centrality

Our Breadth First Traversal plan

- We will build up this algorithm using the GraphBLAS through a series of exercises:
  - Wavefronts and how to move from one wavefront to the next.
  - Iteration across wavefronts
  - Track which vertices have been visited
  - Avoid revisiting vertices
  - Construct the Level Breadth first traversal algorithm

Wavefronts

- A subset of vertices accessed at one stage in a breadth first search pattern … for example ….
  - "You tell two friends and they tell two friends…"

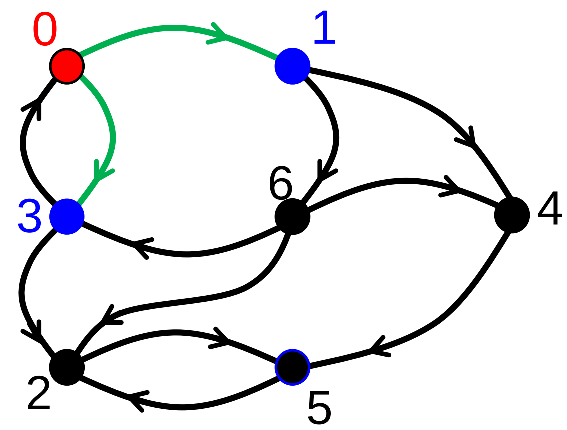


Red=current wavefront and visited, Blue=next wavefront, Black=unvisited

Exercise 7: Traverse the graph

- Modify your code from Exercises 5 to iterate from one wavefront to the next.
- Output each wavefront
- How long before you get a repeating pattern?
  - `GrB_Vector result, vec;`
  - `GrB_Index NODE;`
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - `pretty_print_vector_UINT64(vec, "Input node");`
  - `GrB_Descriptor desc;`
  - `GrB_Descriptor_new(&desc);`
  - `GrB_Descriptor_set(desc, FIELD, VALUE)`
  - `GrB_mxv(result, GrB_NULL, GrB_NULL,`
    `GxB_LOR_LAND_BOOL, graph, vec, desc);`

Solution to exercise 7



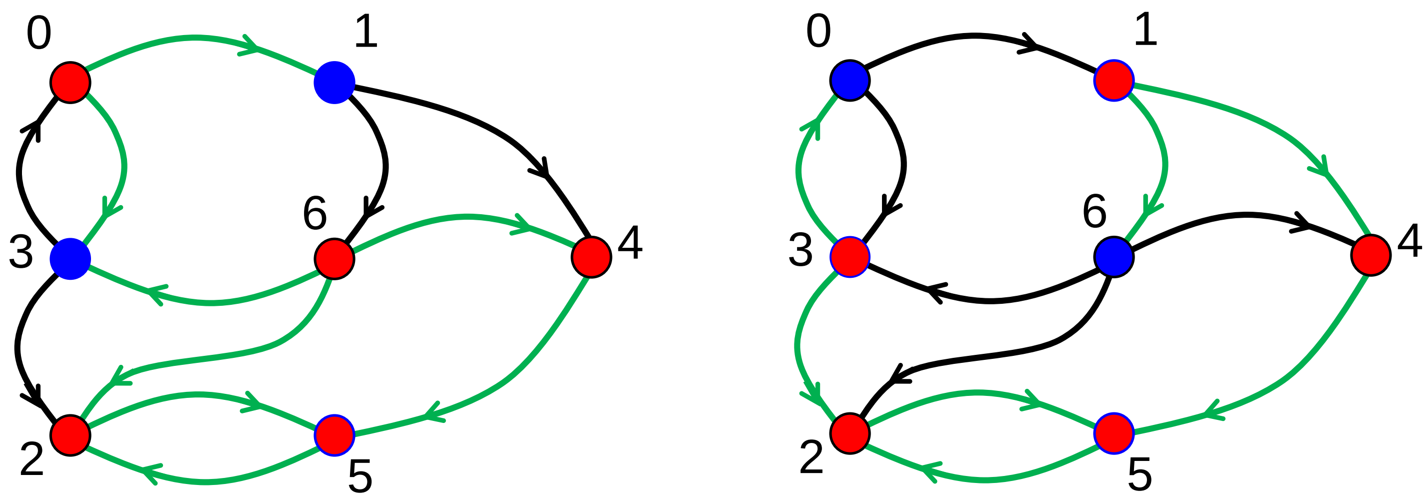

```
// First wavefront has one node set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(w,"wavefront(src)");

for (int i = 0; i < NUM_NODES; ++i) {
  GrB_mxv(w, GrB_NULL, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, w, desc);
  pretty_print_vector_UINT64(w, "wavefront");
}
```

```
Vector: wavefront(src) =
[  1,    -,    -,    -,    -,    -,    -]
Vector: wavefront =
[  -,    1,    -,    1,    -,    -,    -]
Vector: wavefront =
[  1,    -,    1,    -,    1,    -,    1]
Vector: wavefront =
[  -,    1,    1,    1,    1,    1,    -]
Vector: wavefront =
[  1,    -,    1,    -,    1,    1,    1]
Vector: wavefront =
[  -,    1,    1,    1,    1,    1,    -]
Vector: wavefront =
[  1,    -,    1,    -,    1,    1,    1]
…
```

The same container can be used for both input and output
Starts repeating after only a few iterations. Why?

Solution to exercise 7: wavefronts

- “We tell a bunch, and they tell bunch…(rinse and repeat)”



w = {0, 2, 4, 5, 6}

w = {1, 2, 3, 4, 5}

w = {0, 2, 4, 5, 6}

Red=current wavefront and visited, Blue=next wavefront, Black=unvisited

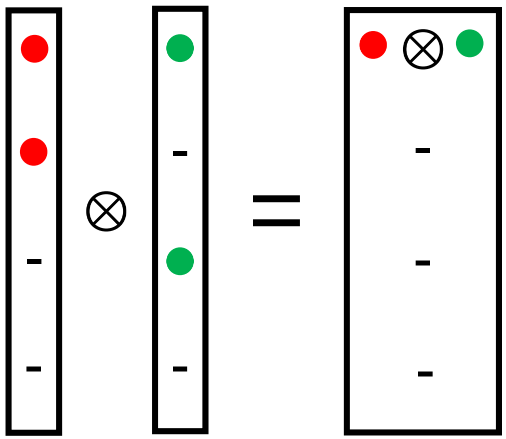
Visited lists

- Breadth-first traversal requires that we visit each node once.
- First step is to keep track of a visited list.
- You can do this by accumulating the wavefronts.
  - Use element-wise logical-OR.

Element-wise Operations: Mult and Add

- $\otimes$ assumes unstored values (-) are the binary operator's ***annihilator***:

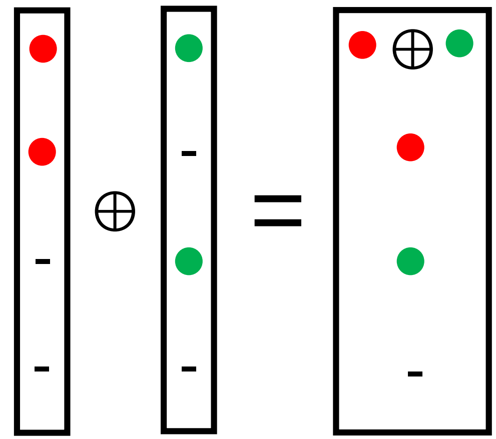
u $\otimes$ **v**



Examples: (x,0), (and, false), (+, ∞)

- $\oplus$ assumes unstored values (-) are the binary operator's ***identity***:

u $\oplus$ **v**



Examples: (+,0), (or, false), (min, ∞)

The rules for element-wise addition also apply to the accumulation operator, $\odot$

GrB_eWiseMult()

$$w \cancel{\otimes} = (u \otimes v)$$

- Compute the element-wise "**multiplication**" of two GraphBLAS vectors.
- Performs the specified operator (op) on the **intersection** of the sparse entries in each input vector , u and v.
  - op could be GrB_BinaryOp, GrB_Monoid, or GrB_Semiring
- Returns error codes of type GrB_info. See the spec for details.

GrB_Info GrB_eWiseMult(GrB_Vector w,
const GrB_Vector ~~mask,~~ → **GrB_NULL**
const GrB_BinaryOp ~~accum,~~ → **GrB_NULL**
const GrB_BinaryOp op,
const GrB_Vector u,
const GrB_Vector v,
const GrB_Descriptor ~~desc);~~ → **GrB_NULL**

Use default values for mask, accum and desc (indicated by GrB_NULL)

GrB_eWiseAdd()

$$w \cancel{\otimes}= (u \oplus v)$$

- Compute the element-wise "**addition**" of two GraphBLAS vectors.
- Performs the specified operator (op) on the **union** of the sparse entries in each input vector , u and v.
  - op could be GrB_BinaryOp, GrB_Monoid, or GrB_Semiring
- Returns error codes of type GrB_info. See the spec for details.

```
GrB_Info GrB_eWiseAdd( GrB_Vector             w,
                       const GrB_Vector       mask,    → GrB_NULL
                       const GrB_BinaryOp     accum,   → GrB_NULL
                       const GrB_BinaryOp     op,
                       const GrB_Vector       u,
                       const GrB_Vector       v,
                       const GrB_Descriptor   desc);   → GrB_NULL
```

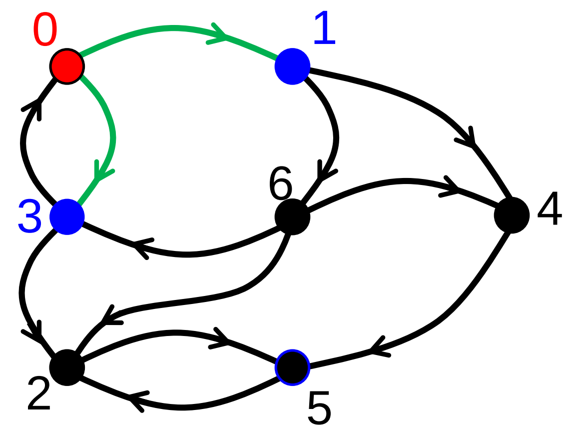
Use default values for mask, accum and desc (indicated by GrB_NULL)

Exercise 8: Keep track of ‘visited’ nodes

- Modify code from Exercise 7 to compute the visited set as you iterate.
  - `GrB_Vector result, vec;`
  - `GrB_Index NODE;`
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - `pretty_print_vector_UINT64(vec, ”Input node");`
  - `GrB_Descriptor desc;`
  - `GrB_Descriptor_new(&desc);`
  - `GrB_Descriptor_set(desc, ARG, OP)`
  - ```
    GrB_eWiseAdd(vec, GrB_NULL, GrB_NULL,
                 GrB_LOR, vec, wav, GrB_NULL);
    ```
  - ```
    GrB_mxv(result, GrB_NULL, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, vec, desc);
    ```

Solution to exercise 8

```
// First wavefront has node 0 set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w, v;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&v, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);

pretty_print_vector_UINT64(w, "wavefront(src)");

for (int i=0; i<NUM_NODES; ++i) {
  GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
               GrB_LOR, v, w, GrB_NULL);
  pretty_print_vector_UINT64(v, "visited");
  GrB_mxv(w, GrB_NULL, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, w, desc);
  pretty_print_vector_UINT64(w, "wavefront");
}
```

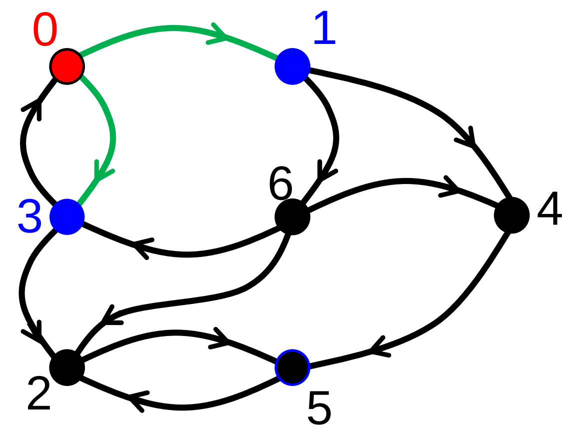



```
Vector: wavefront(src) =
[  1,    -,    -,    -,    -,    -,    -]

Vector: visited =
[  1,    -,    -,    -,    -,    -,    -]
Vector: wavefront =
[  -,    1,    -,    1,    -,    -,    -]

Vector: visited =
[  1,    1,    -,    1,    -,    -,    -]
Vector: wavefront =
[  1,    -,    1,    -,    1,    -,    1]

Vector: visited =
[  1,    1,    1,    1,    1,    -,    1]
Vector: wavefront =
[  -,    1,    1,    1,    1,    1,    -]

Vector: visited =
[  1,    1,    1,    1,    1,    1,    1]
Vector: wavefront =
[  1,    -,    1,    -,    1,    1,    1]

Vector: visited =
[  1,    1,    1,    1,    1,    1,    1]
…
```

Solution to exercise 8

```
// First wavefront has node 0 set.
GrB_Index const SRC_NODE = 0;
GrB_Vector w, v;
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_new(&v, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc

pretty_print_vector_UIN

for (int i=0; i<NUM_NODES; ++i) {
  GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
               GrB_LOR, v, w, GrB_NULL);
  pretty_print_vector_UINT64(v, "visited");
  GrB_mxv(w, GrB_NULL, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, w, tran);
  pretty_print_vector_UINT64(w, "wavefront");
}
```

What should the exit condition be?




```
Vector: wavefront(src) =
[  1,   -,   -,   -,   -,   -,   -]

Vector: visited =
[  1,   -,   -,   -,   -,   -,   -]
Vector: wavefront =
   -,   1,   -,   1,   -,   -,   -]

ctor: visited =
   1,   1,   -,   1,   -,   -,   -]
ctor: wavefront =
   1,   -,   1,   -,   1,   -,   1]

Vector: visited =
[  1,   1,   1,   1,   1,   -,   1]
Vector: wavefront =
[  -,   1,   1,   1,   1,   1,   -]

Vector: visited =
[  1,   1,   1,   1,   1,   1,   1]
Vector: wavefront =
[  1,   -,   1,   -,   1,   1,   1]

Vector: visited =
[  1,   1,   1,   1,   1,   1,   1]
…
```

GrB_mxv()

$$w\langle \neg \mathbf{m}, z\rangle \cancel{\otimes}= (A \oplus . \otimes u)$$

- …say something
- Say something else....

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

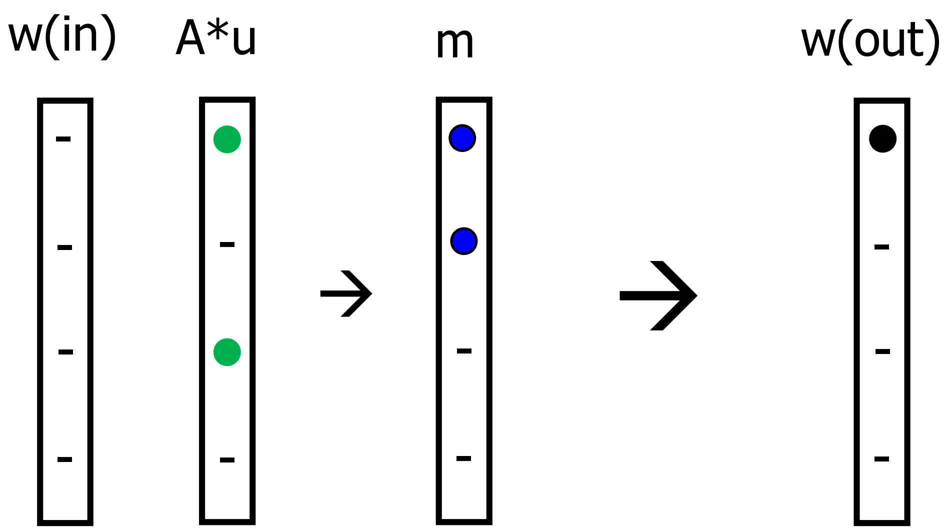
GrB_NULL

It's time to explain masking and REPLACE in GraphBLAS operations.

Masking

- Every GraphBLAS operation that computes an opaque matrix or vector supports a “write mask”
- A mask, m, controls which elements of the output can be written:
  - Same size as output object (mask vectors or mask matrices)
  - Any location in the mask that evaluates to ‘true’ can be written in the output object

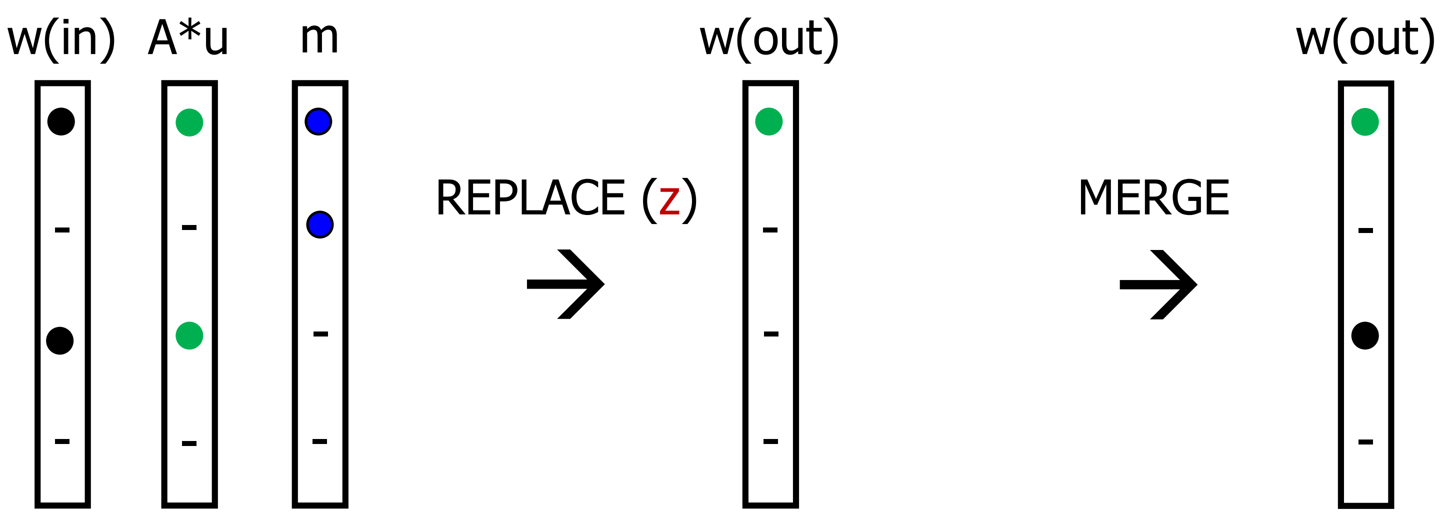
$$w\langle \mathbf{m} \rangle = (A \oplus . \otimes u)$$



REPLACE vs. "MERGE"

- When a mask is used and the output container is not empty when the operation is called…what do you do to the "masked out" elements?
  - REPLACE (z): all unwritten locations are cleared (zeroed out).
  - MERGE: all unwritten locations are left alone.
- Behaviour defaults to MERGE; otherwise, use a descriptor:
  - GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE)

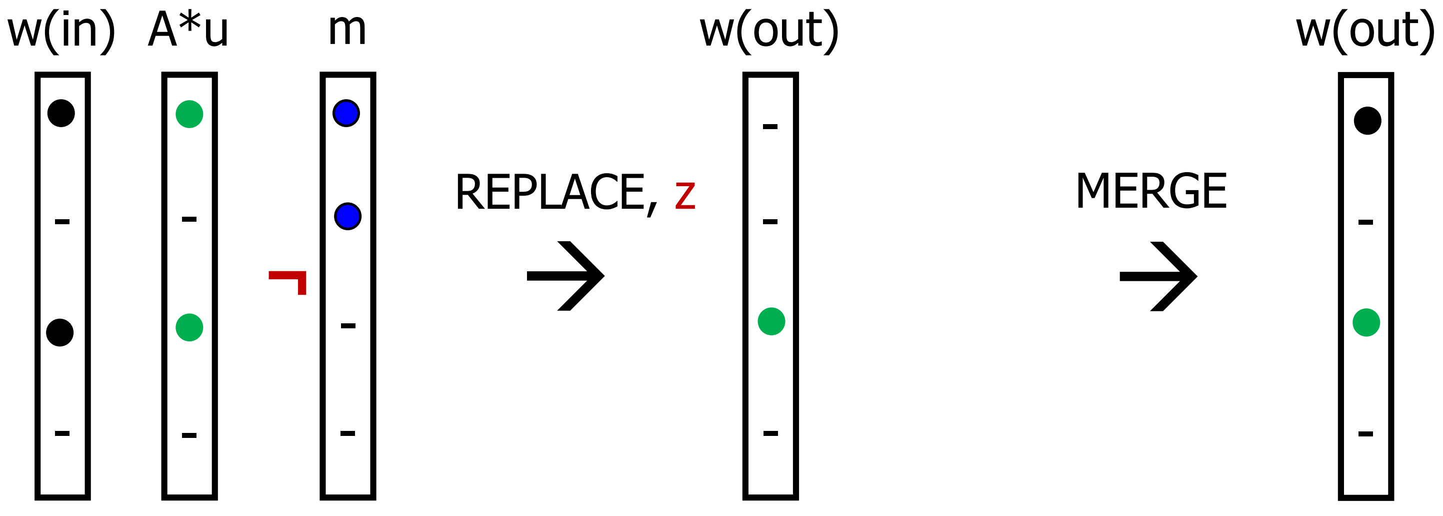
$$w\langle \mathbf{m}, z\rangle = (A \oplus . \otimes u)$$



Structural Complement (mask)

- Specified with a descriptor:
  - GrB_Descriptor_set(desc, GrB_MASK, GrB_SCMP)
- Inverts the logic of mask (write enabled on false)
- A mask, m, is interpreted as a logical ‘stencil’ that controls which elements of the output can be written:
  - Any location in the mask that evaluates to ‘true’ can be written

$$w\langle \neg m, z\rangle = (A \oplus . \otimes u)$$



Using Descriptors (summary)

- A descriptor *field* selects the object it impacts:
  - **GrB_INP0**: The first input GraphBLAS object
  - **GrB_INP1**: The second input GraphBLAS object
  - **GrB_MASK**: The GraphBLAS mask object
  - **GrB_OUTP**: The output GraphBLAS object

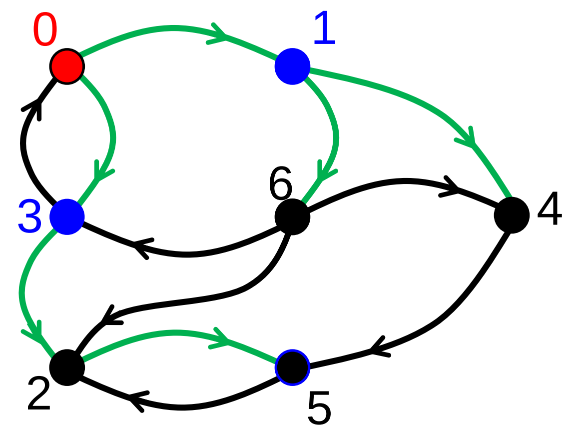
- Each field supports one *value* (currently):
  - **GrB_INP0**: **GrB_TRAN** (transpose)
  - **GrB_INP1**: **GrB_TRAN** (transpose)
  - **GrB_MASK**: **GrB_SCMP** (structural complement)
  - **GrB_OUTP**: **GrB_REPLACE** (clear the output before writing result)

Exercise 9: Avoid revisiting

- Use the visited list as a mask prevent revisiting previous nodes
- Exit the loop when there is no more 'work' to be done
- You will need the following types and methods from the GraphBLAS
  - `GrB_Vector_new(&vec, GrB_BOOL, NUM_NODES);`
  - `GrB_Vector_setElement(vec, true, NODE);`
  - ```
    GrB_eWiseAdd(vec, GrB_NULL, GrB_NULL,
                 GrB_LOR, vec, wav, GrB_NULL);
    ```
  - ```
    GrB_mxv(result, GrB_NULL, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, vec, desc);
    ```
  - `GrB_Descriptor desc;`
  - `GrB_Descriptor_new(&desc);`
  - `GrB_Descriptor_set(desc, FIELD, VALUE)`

```
FIELD: GrB_INP0, GrB_INP1, GrB_OUTP, GrB_MASK
VALUE: GrB_TRAN, GrB_REPLACE, GrB_SCMP
```

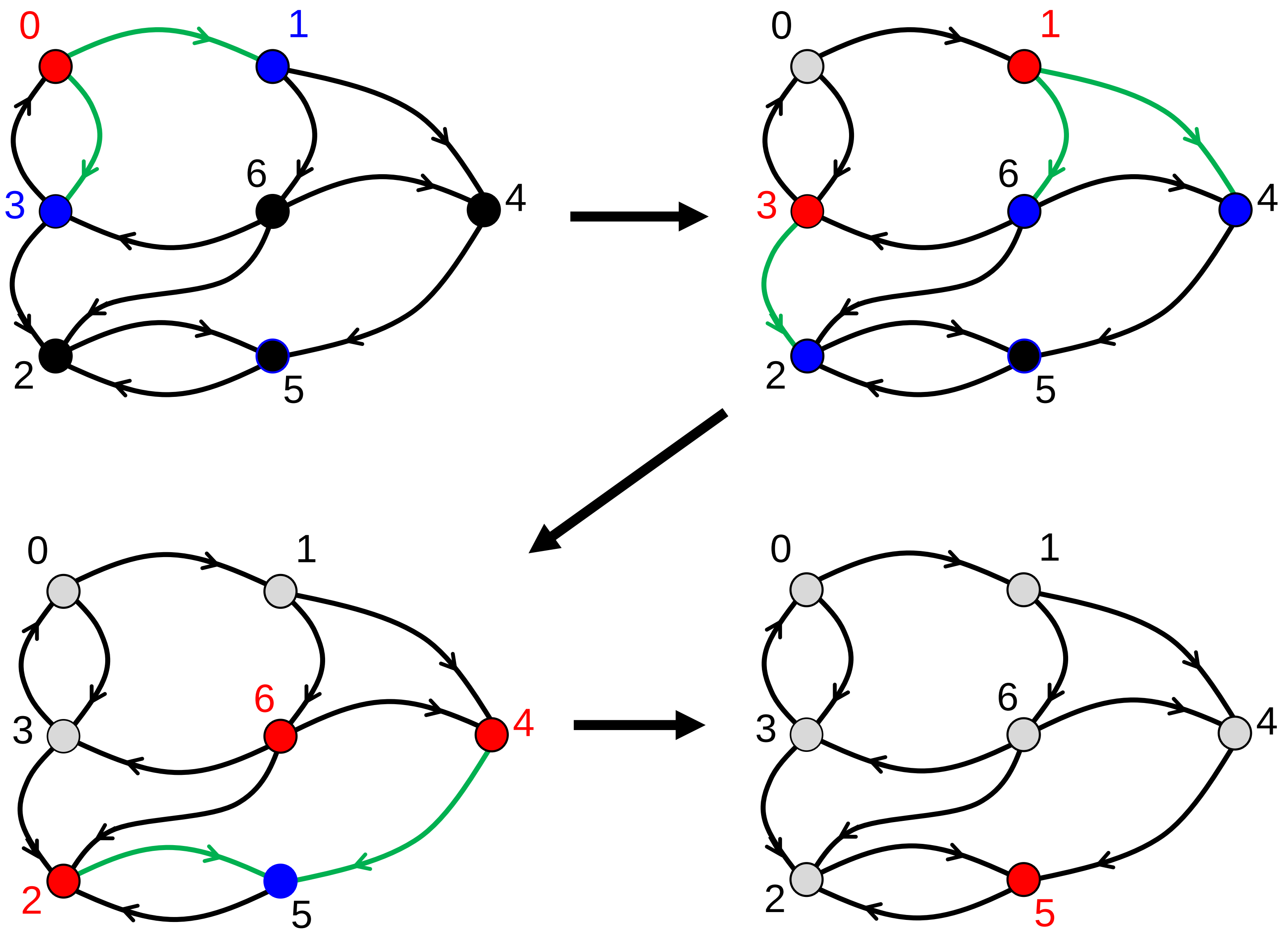
Solution to exercise 9




```
...
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);
GrB_Descriptor_set(desc, GrB_MASK, GrB_SCMP);
GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE);

pretty_print_vector_UINT64(w, "wavefront(src)");

GrB_Index nvals = 0;
do {
  GrB_eWiseAdd(v, GrB_NULL, GrB_NULL,
                 GrB_LOR, v, w, GrB_NULL);
  pretty_print_vector_UINT64(v, "visited");
  GrB_mxv(w, v, GrB_NULL,
            GxB_LOR_LAND_BOOL, graph, w, desc);
  pretty_print_vector_UINT64(w, "wavefront");
  GrB_Vector_nvals(&nvals, w);
} while (nvals > 0);
```

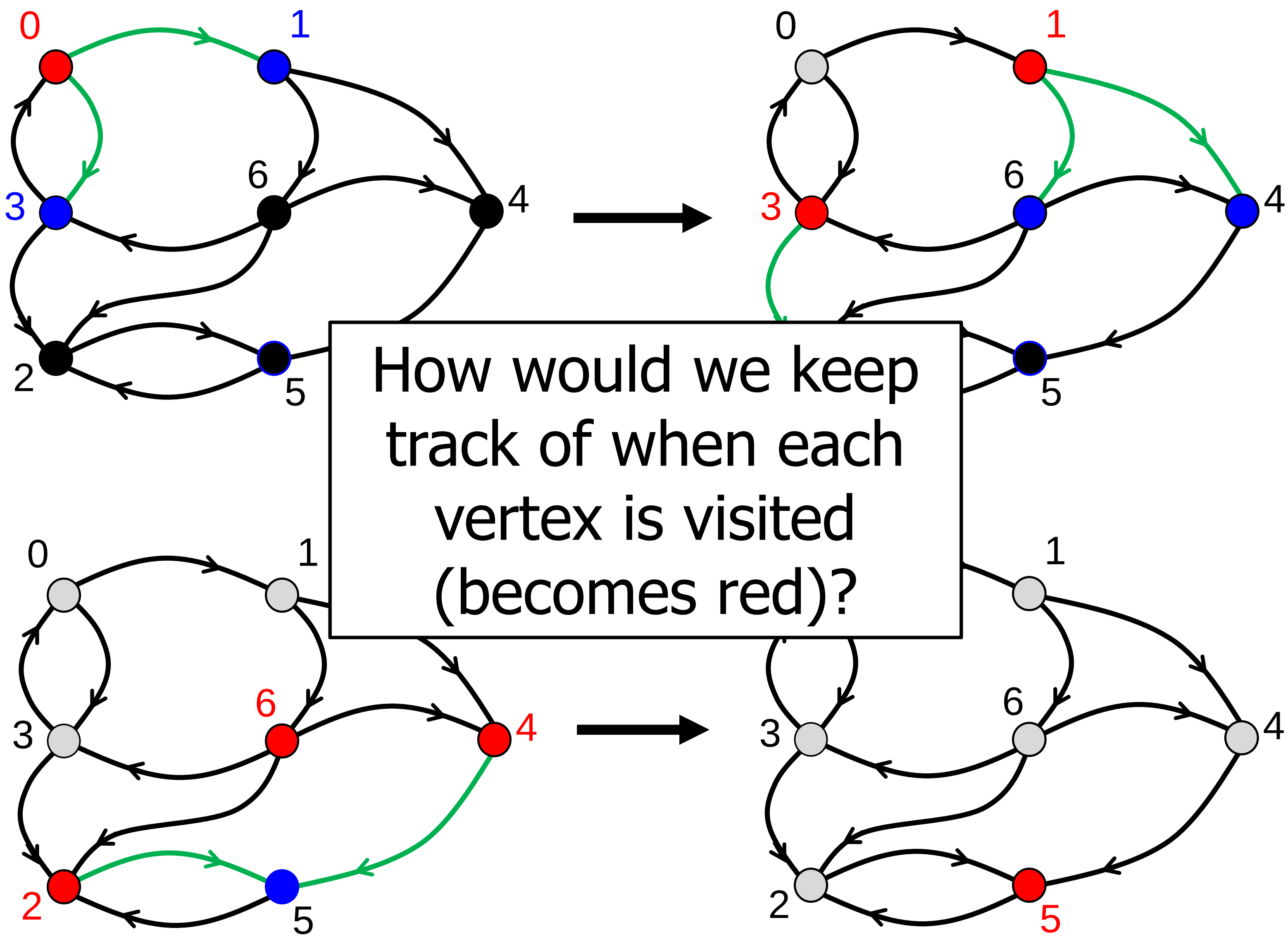
```
Vector: wavefront(src) =
[  1,   -,   -,   -,   -,   -,   -

Vector: visited =
[  1,   -,   -,   -,   -,   -,   -]
Vector: wavefront =
[  -,   1,   -,   1,   -,   -,   -]

Vector: visited =
[  1,   1,   -,   1,   -,   -,   -]
Vector: wavefront =
[  -,   -,   1,   -,   1,   -,   1]

Vector: visited =
[  1,   1,   1,   1,   1,   -,   1]
Vector: wavefront =
[  -,   -,   -,   -,   -,   1,   -]

Vector: visited =
[  1,   1,   1,   1,   1,   1,   1]
Vector: wavefront =
[  -,   -,   -,   -,   -,   -,   -]
```

Breadth-First Traversal



Red=current wavefront and visited, Blue=next wavefront, Gray=visited, Black=unvisited

Breadth-First Traversal



How would we keep track of when each vertex is visited (becomes red)?

Red=current wavefront and visited, Blue=next wavefront, Gray=visited, Black=unvisited

GrB_assign() from constant $\mathbf{w}(\boldsymbol{i}) \odot = c$

- Assign a constant to a subset of the output vector.
- Locations to be assigned selected by an output index vector, indices:

```
w(indices[j]) = c,                         ∀ j : 0 ≤ j < nindices,
w(indices[j]) = w(indices[j]) ⊙ c,         ∀ j : 0 ≤ j < nindices.
```

```
GrB_Info GrB_assign( GrB_Vector              w,
                     const GrB_Vector        mask,
                     const GrB_BinaryOp      accum,
                     const GrB_Vector        u,
                     const GrB_Index        *indices,
                     const GrB_Index         nindices,
                     const GrB_Descriptor    desc);
```

- Use a constant **`GrB_ALL`** in place of the **`indices`** argument to select that all elements of **`w`** are to be assigned to (in order 0 to 1-nindices).

GrB_assign()

- There are several variants of assign
  - Standard vector assignment
  - Standard matrix assignment

$$\mathbf{w}(\boldsymbol{i}) \odot= \mathbf{u} \qquad\qquad \mathbf{C}(\boldsymbol{i}, \boldsymbol{j}) \odot= \mathbf{A}$$

  - Assign a vector to the elements of column c_j of a matrix
  - Assign a vector to the elements of row r_i of a matrix

$$\mathbf{C}(\boldsymbol{i}, cj) \odot= \mathbf{u} \qquad\qquad \mathbf{C}(r_i, \boldsymbol{j}) \odot= \mathbf{u}^{\mathrm{T}}$$

  - Assign a constant to a subset of a vector.
  - Assign a constant to a subset of a matrix.

$$\mathbf{w}(\boldsymbol{i}) \odot= c \qquad\qquad \mathbf{C}(\boldsymbol{i}, \boldsymbol{j}) \odot= c$$

A and **C** are GraphBLAS matrices. **u** and **w** are GraphBLAS vectors **i** and **j** are index vectors

GrB_assign() from vector $\mathbf{w}(\boldsymbol{i}) \odot= \mathbf{u}$

- Assign a vector to a subset of the output vector.
- Values to be assigned selected by an output index vector, *i*

```
w(indices[j]) = u(j),                    ∀ j : 0 ≤ j < nindices,
w(indices[j]) = w(indices[j]) ⊙ u(j), ∀ j : 0 ≤ j < nindices.
```

```
GrB_Info GrB_assign( GrB_Vector                   w,
                     const GrB_Vector             mask,
                     const GrB_BinaryOp           accum,
                     const GrB_Vector             u,
                     const GrB_Index             *indices,
                     const GrB_Index              nindices,
                     const GrB_Descriptor         desc);
```

- Use a constant **`GrB_ALL`** in place of the **`indices`** argument to select that all elements of u are to be assigned in order to **`w`**.

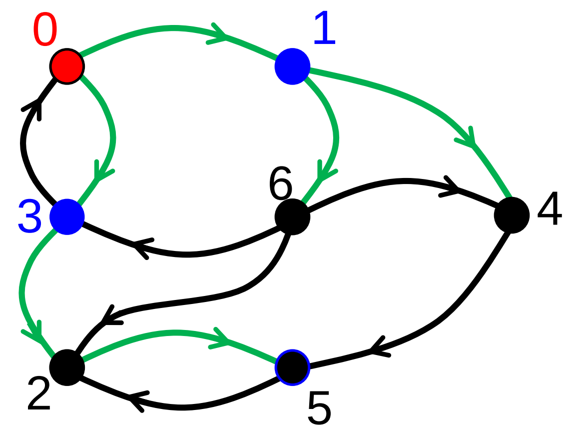
Exercise 10: level BFS

- Modify the code from Exercise 9 to compute the level at which each node is encountered:
  - SRC_NODE is level 1, Its neighbors are level 2, … and so forth
- Challenge: use assign in place of eWiseAdd

```
– GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
– GrB_Vector_setElement(w, true, SRC_NODE);
– GrB_Descriptor desc;
– GrB_Descriptor_new(&desc);
– GrB_Descriptor_set(desc, FIELD, VALUE);
– pretty_print_vector_UINT64(vec, "levels");
– GrB_assign(u, mask, accum, c, GrB_ALL, NUM_NODES, desc);
– GrB_mxv(w, mask, accum, GxB_LOR_LAND_BOOL, graph, w, desc);
– GrB_Vector_nvals(&nvals, w);
```

```
FIELD: GrB_INP0, GrB_INP1, GrB_OUTP, GrB_MASK
VALUE: GrB_TRAN, GrB_REPLACE, GrB_SCMP
```

Solution to exercise 10




```
GrB_Vector_new(&levels, GrB_UINT64, NUM_NODES);
GrB_Vector_new(&w, GrB_BOOL, NUM_NODES);
GrB_Vector_setElement(w, true, SRC_NODE);

GrB_Descriptor desc;
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, ...);

pretty_print_vector_BOOL(w, "wavefront(src)");
GrB_Index nvals = 0, lvl = 0;
do {
  ++lvl;
  GrB_assign(levels, w, GrB_NULL,
             lvl, GrB_ALL, NUM_NODES, GrB_NULL);
  pretty_print_vector_UINT64(levels, "levels");

  GrB_mxv(w, levels, GrB_NULL,
          GxB_LOR_LAND_BOOL, graph, w, desc);
  pretty_print_vector_BOOL(w, "wavefront");
  GrB_Vector_nvals(&nvals, w);
} while (nvals > 0);
```

```
Vector: wavefront(src) =
[  1,   -,   -,   -,   -,   -,   -]

Vector: levels =
[  1,   -,   -,   -,   -,   -,   -]
Vector: wavefront =
[  -,   1,   -,   1,   -,   -,   -]

Vector: levels =
[  1,   2,   -,   2,   -,   -,   -]
Vector: wavefront =
[  -,   -,   1,   -,   1,   -,   1]

Vector: levels =
[  1,   2,   3,   2,   3,   -,   3]
Vector: wavefront =
[  -,   -,   -,   -,   -,   1,   -]

Vector: levels =
[  1,   2,   3,   2,   3,   4,   3]
Vector: wavefront =
[  -,   -,   -,   -,   -,   -,   -]
```

The GraphBLAS Operations

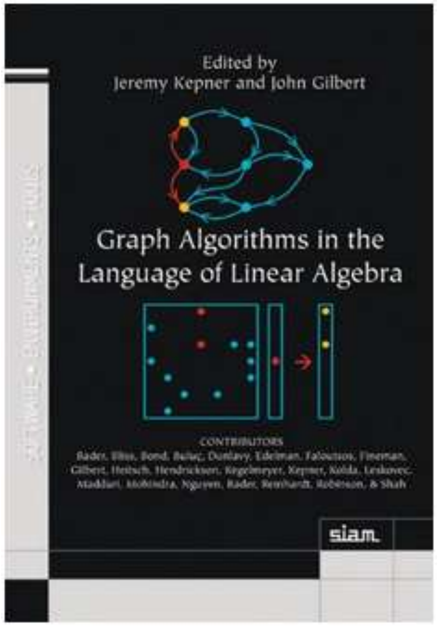
Operation Name	Mathematical Notation		
mxm	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot \mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T, z\rangle$	$=$	$\mathbf{w}^T \odot \mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot [\oplus_j \mathbf{A}(:, j)]$
reduce (scalar)	s	$=$	$s \odot [\oplus_{i,j} \mathbf{A}(i, j)]$
	s	$=$	$s \odot [\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot f_u(\mathbf{u})$
transpose	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot \mathbf{A}^T$
extract	$\mathbf{C}\langle\mathbf{M}, z\rangle$	$=$	$\mathbf{C} \odot \mathbf{A}(\boldsymbol{i}, \boldsymbol{j})$
	$\mathbf{w}\langle\mathbf{m}, z\rangle$	$=$	$\mathbf{w} \odot \mathbf{u}(\boldsymbol{i})$
assign	$\mathbf{C}\langle\mathbf{M}, z\rangle(\boldsymbol{i}, \boldsymbol{j})$	$=$	$\mathbf{C}(\boldsymbol{i}, \boldsymbol{j}) \odot \mathbf{A}$
	$\mathbf{w}\langle\mathbf{m}, z\rangle(\boldsymbol{i})$	$=$	$\mathbf{w}(\boldsymbol{i}) \odot \mathbf{u}$

We've covered only a small fraction of the GraphBLAS Operations

The same conventions are used across all operations so the operations we did not cover are straightforward to pick up

Conclusion and next steps

- The GraphBLAS define a standard API for "Graph Algorithms in the Language of Linear Algebra".
- A wide range of algorithms are variations of the basic breadth first traversal for a graph.
- To reach GraphBLAS mastery
  - Attend the Graph Algorithms Building Blocks workshop at IPDPS
  - Explore the challenge problems included with this tutorial
  - Work through the algorithms in the Graph book →



GraphBLAS at HPEC 2018

- GraphBLAS is a community effort. Join the community:
  - Go to graphblas.org and join our mailing list

- Attend the HPEC GraphBLAS Birds of a Feather (BOF) 6 PM to 7 PM, Eden Vale C1/C2.

- Please send us feedback about the tutorial

  timothy.g.mattson@intel.com

  smcmillan@sei.cmu.edu

  - Tell us what you really liked.
  - Tell us what we should change
  - Tell us what you wish we'd covered but didn't
  - Plus anything else that might help us improve

Appendices

- MxM: the low-level details of the GraphBLAS operations
- Challenge Problems: Some key algorithms with the GraphBLAS
- SuiteSparse: usage notes, extensions and future plans
- Reference material

GraphBLAS: details of operations

- When you read the GraphBLAS C API specification, the operations are described in a manner that may seem obtuse.
- The definitions, however, are presented in this way for good reasons:
  - to cover the full range of variations exposed by the various arguments and to express the operation without ever specifying the undefined elements (i.e. the “zeros” of the semiring).
  - To avoid any reference to the non-stored elements of the sparse matrix. In sparse arrays, the undefined elements are usually assumed to be the ”zero of the semiring”. By defining the operations without any reference to those “un-stored values”, we can freely change the semirings between operations without having to update the un-stored elements.

GrB_mxm()

$$\mathbf{C} = \mathbf{A} \oplus . \otimes \mathbf{B} = \mathbf{A}\mathbf{B}$$

Matrix Multiplication … the way we learned it in school

$$\mathbf{C}(i,j) = \bigoplus_{k=1}^{l} \mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$$

$$\mathbf{A} : \mathbb{S}^{m \times l} \qquad \mathbf{B} : \mathbb{S}^{l \times n} \qquad \mathbf{C} : \mathbb{S}^{m \times n}$$

Matrix Multiplication … set notation to ignore un-stored elements

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

With set notation, it's easier to define the operations over a matrix as the semi-ring changes

GrB_mxm(): Function Signature

```
GrB_Info GrB_mxm(GrB_Matrix                 *C,
                 const GrB_Matrix            Mask,
                 const GrB_BinaryOp          accum,
                 const GrB_Semiring          op,
                 const GrB_Matrix            A,
                 const GrB_Matrix            B,
                 const GrB_Descriptor        desc);
```

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

Mask (IN) A "write" mask that controls which results from this operation are stored into the output matrix C (optional). If no mask is desired, GrB_NULL is specified. The Mask dimensions must match those of the matrix C and the domain of the Mask matrix must be of type bool or any "built-in" GraphBLAS type.

accum (IN) A binary operator used for accumulating entries into existing C entries. For assignment rather than accumulation, GrB_NULL is specified.

op (IN) Semiring used in the matrix-matrix multiply: op $= \langle D_1, D_2, D_3, \oplus, \otimes, 0\rangle$.

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

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

desc (IN) Operation descriptor (optional). If a *default* descriptor is desired, GrB_NULL should be used. Valid fields are as follows:

Argument	Field	Value	Description
C	GrB_OUTP	GrB_REPLACE	Output matrix C is cleared (all elements removed) before result is stored in it.
Mask	GrB_MASK	GrB_SCMP	Use the structural complement of Mask.
A	GrB_INP0	GrB_TRAN	Use transpose of A for operation.
B	GrB_INP1	GrB_TRAN	Use transpose of B for operation.

GrB_mxm(): Function Signature

```
GrB_Info GrB_mxm(GrB_Matrix                    *C,
                 const GrB_Matrix               Mask,
                 const GrB_BinaryOp             accum,
                 const GrB_Semiring             op,
                 const GrB_Matrix               A,
                 const GrB_Matrix               B,
                 const GrB_Descriptor           desc);
```

GrB_Info return values:

GrB_SUCCESS	Blocking mode: Operations completed successfully. Nonblocking mode: consistency tests passed on dimensions and domains for input arguments
GrB_PANIC	Unknown Internal error
GrB_OUTOFMEM	Not enough memory for the operation
GrB_DIMENSION_MISMATCH	Matrix dimensions are incompatible.
GrB_DOMAIN_MISMATCH	Domains of matrices are incompatible with the domains of the accumulator, semiring, or mask.

Standard function behavior

- Consider the following code:

```
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE);
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRANS);
GrB_mxm(&C, M, Int32Add, Int32AddMul, A, B, desc);
```

int32AddMul semiring
int32Add accumulation

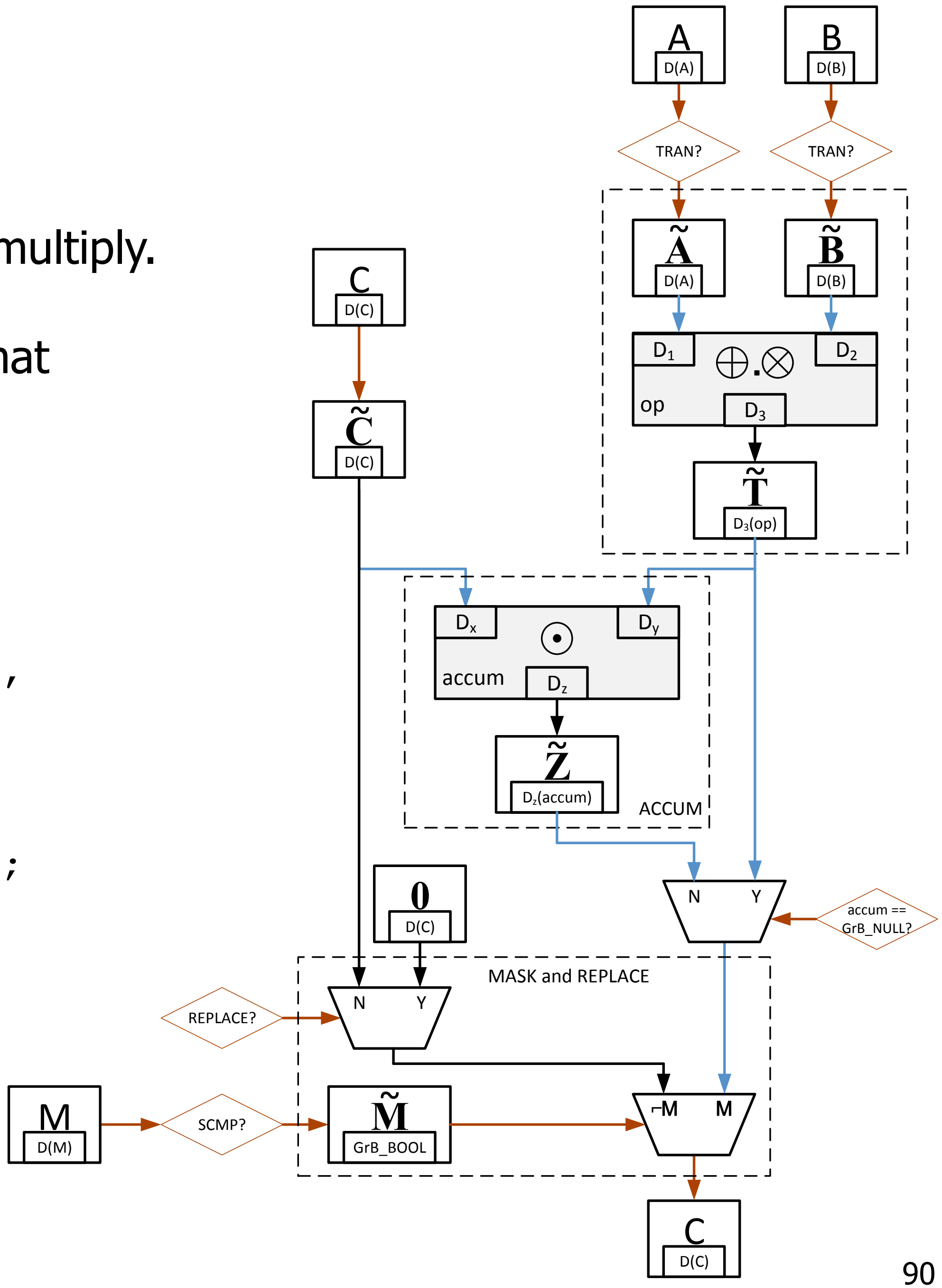
Form input operands and mask based on descriptor	C, B, M, A $\leftarrow$ A^T
Test the domains and sizes for consistency.	int32, dims match
Carry out the indicated operation	T $\leftarrow$ A *.+ B, Z $\leftarrow$ C + T
Apply the write-mask to select output values	Z $\leftarrow$ Z $\cap$ M
Replace mode: delete elements in output object and replace with output values	C $\leftarrow$ Z
Merge mode: Assign output value (i,j) to element (i,j) of output object, but leave other elements of the output object alone.	

MXM flowchart

To understand what happens inside a graphBLAS operation, consider matrix multiply.

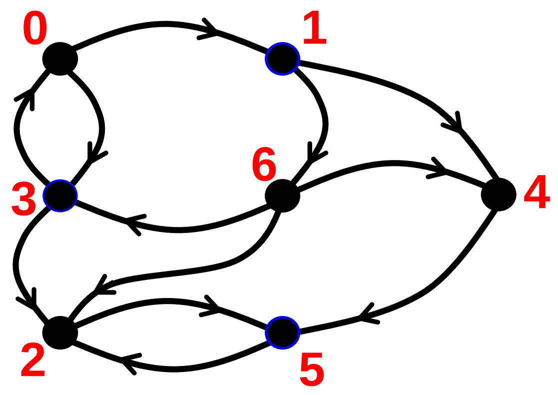
All the operations follow this basic format

```
GrB_Info GrB_mxm(
    GrB_Matrix                C,
    const GrB_Matrix          M,
    const GrB_BinaryOp        accum,
    const GrB_Semiring        op,
    const GrB_Matrix          A,
    const GrB_Matrix          B,
    const GrB_Descriptor      desc);
```



Exercise: Matrix Matrix Multiplication

- Multiply the adjacency matrix from our “logo graph” by itself.
- Print resulting matrix and interpret the result
- Hint: Do the multiply again and compare results. Do you see the pattern?



Appendices

- MxM: the low-level details of the GraphBLAS operations
- Challenge Problems: Some key algorithms with the GraphBLAS
- SuiteSparse: usage notes, extensions and future plans
- Reference material

Challenge problems

- Triangle counting
- PageRank
- Betweenness Centrality
- Maximal Independent Set

Work in Progress: We should make a slide for each problem defining the algorithm in enough detail so students can implement the GraphBLAS implementation on their own

Counting Triangles (once) with GraphBLAS

- Given:
  - Undirected graph G = {V, E}
  - L: boolean, lower-triangular portion of adjacency matrix
- **# triangles = $\|L \otimes (L \oplus.\otimes L^T)\|_1$**
  - Semiring can be Plus-AND or Plus-Times
  - Element-wise multiplication is equivalent to a mask operation

```
uint64_t triangle_count(GrB_Matrix L)   // L: NxN, lower-triangular, boolean
{
    GrB_Index N;
    GrB_Matrix_nrows(&N, L);
    GrB_matrix C;
    GrB_Matrix_new(&C, GrB_UINT64, N, N);

    GrB_mxm(C, L, GrB_NULL, GrB_UInt64AddMul, L, L, GrB_TB); // C<L> = L * L^T

    uint64_t count;
    GrB_reduce(&count, GrB_NULL, GrB_UInt64Add, C, GrB_NULL);// 1-norm of C
    return count;
}
```

Appendices

- MxM: the low-level details of the GraphBLAS operations
- Challenge Problems: Some key algorithms with the GraphBLAS
- SuiteSparse: usage notes, extensions and future plans
- Reference material

SuiteSparse:GraphBLAS

- Full implementation of GraphBLAS Specification written by Tim Davis, Texas A&M University
- Easy-to-read User Guide with lots of examples
- Already in Ubuntu, Debian, Mac HomeBrew, ...
- Most operations just as fast as MATLAB (like C=A*B)
- assign and setElement can be 1000x faster (or more!) than MATLAB, by exploiting non-blocking mode
- V2.1: matrices by-row and by-column; by-row is often faster than by-column when A(i,j) is the edge (i,j). Compile with –DBYROW or use GxB_set(…)
- Graph algorithms in GraphBLAS typically faster than novice-level graph algorithm without GraphBLAS, and easier to write
- **`http://faculty.cse.tamu.edu/davis/GraphBLAS`**

SuiteSparse:GraphBLAS extensions

- MATLAB-like colon notation for GrB_assign, extract
- unary operators ONE, ABS, LNOT_[type]
- ISEQ, ISNE, ISLT, ... return same type as inputs (e.g. PLUS monoid cannot be combined with Boolean EQ, but PLUS-ISEQ can, to count the number of equal pairs)
- query: size of type, type of matrix, ...
- GxB_select: like MATLAB L=tril(A,k), d=diag(A), ...
- GxB_get/set: to change matrix format (by row, by col, hypersparse)
- 44 built-in monoids
- 960 built-in semirings (like GxB_LOR_LAND_BOOL)
- GxB_resize: change size of matrix or vector
- GxB_subassign: variation of GrB_assign
- GxB_kron: Kronecker product
- Thread-safe if called by user application threads, in parallel

SuiteSparse:GraphBLAS future

- Multicore parallelism via OpenMP
- Variable-sized types (imagine matrix of matrices, or a matrix of arbitrary-sized integers with 10's or 1000's of digits)
- Solvers: Ax=b over a group (double, GF(2), ...)
- Better performance: e.g. many monoids could terminate quickly:
  - OR (x1, x2, x3, ...) becomes true as soon as any xi = true
  - also for AND, and reduction ops FIRST and SECOND
- Iterators for algorithms like depth-first-search
- Reduction to vector or scalar: could also return the index for some operators (MAX, MIN, FIRST, SECOND): argmin, argmax
- Pretty-print methods
- Serialization to/from a binary string: for binary file I/O, or sending/receiving a GrB_Matrix in an MPI message; with compression
- Priority queue: a GrB_Vector acting like a heap
- Concatenate: like C=[A;B] in MATLAB
- Interface to MATLAB, Julia, Python, ...
- Faster C=A*B for user-defined types and operators

Appendices

- MxM: the low-level details of the GraphBLAS operations
- Challenge Problems: Some key algorithms with the GraphBLAS
- SuiteSparse: usage notes, extensions and future plans
- Reference material

Full set of GraphBLAS opaque objects

Table 2.1: GraphBLAS opaque objects and their types.

GrB_Object types	Description
GrB_Type	User-defined scalar type.
GrB_UnaryOp	Unary operator, built-in or associated with a single-argument C function.
GrB_BinaryOp	Binary operator, built-in or associated with a two-argument C function.
GrB_Monoid	Monoid algebraic structure.
GrB_Semiring	A GraphBLAS semiring algebraic structure.
GrB_Matrix	Two-dimensional collection of elements; typically sparse.
GrB_Vector	One-dimensional collection of elements.
GrB_Descriptor	Descriptor object, used to modify behavior of methods.

Error codes returned by GraphBLAS methods
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.

Error codes returned by GraphBLAS methods
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.