

# 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

Copyright 2019 Carnegie Mellon University and Intel Corporation.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM19-0946

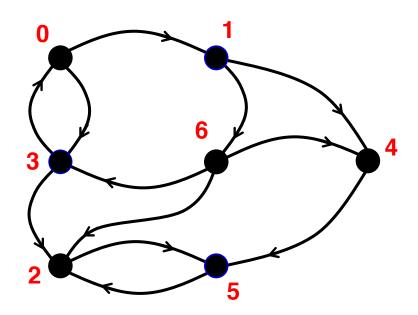
#### **Outline**



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

## Understanding relationships between items

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

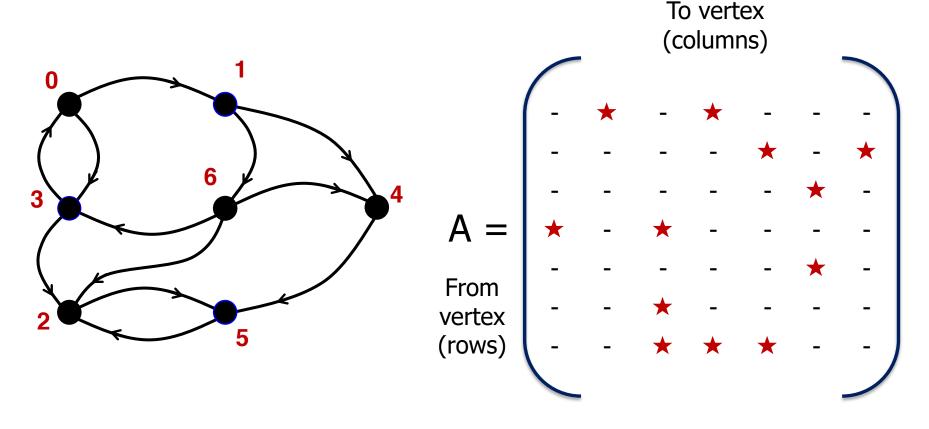


• Graph: Two sets, one for the vertices (v) and one for the edges (e)  $v \in [0, 1, 2, 3, 4, 5, 6]$ 

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

## A graph as a matrix

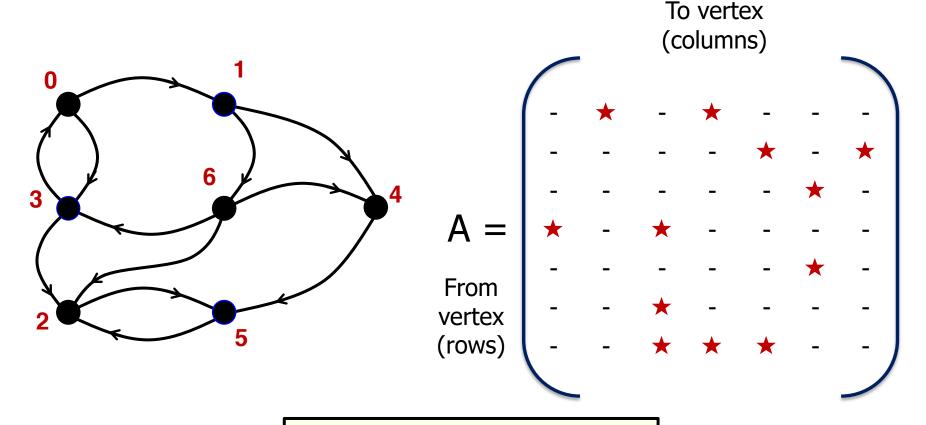
 Adjacency Matrix: A square matrix (usually sparse) where rows and columns are labeled by vertices and non-empty values are edges from a row vertex to a column vertex



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

## A <u>Directed</u> graph as a matrix

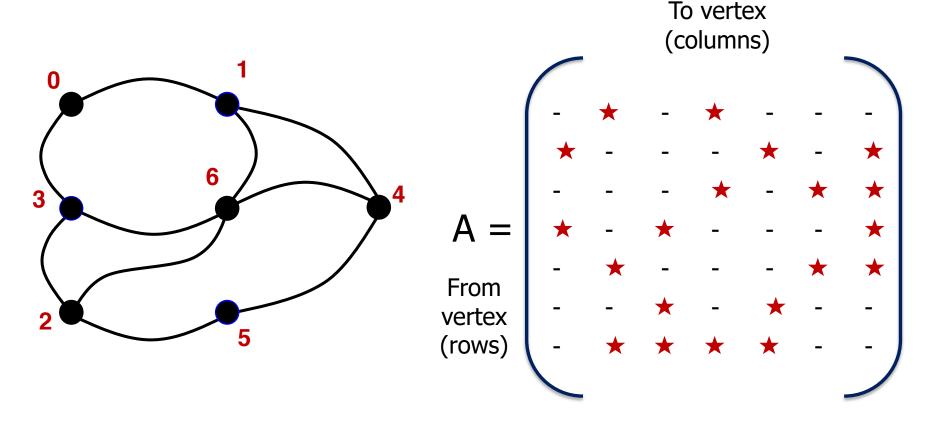
 Adjacency Matrix: A square matrix (usually sparse) where rows and columns are labeled by vertices and non-empty values are edges from a row vertex to a column vertex



This is a directed graph (the edges have arrows)

## An Undirected graph as a matrix

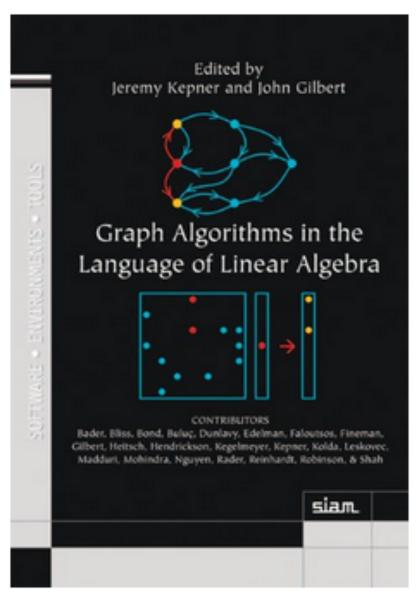
 Adjacency Matrix: A square matrix (usually sparse) where rows and columns are labeled by vertices and non-empty values are edges from a row vertex to a column vertex



This is an undirected graph (no arrows on the edges) and the Adjacency matrix is symmetric

## **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?

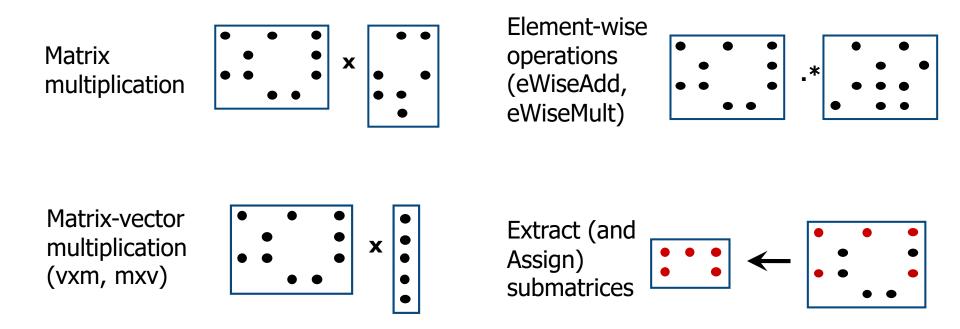
- They do so in terms of the BLAS:
  - The Basic Linear Algebra Subprograms: low-level building blocks from which any linear algebra algorithm can be written

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



...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<sup>†</sup>, Tim Mattson<sup>‡</sup>, Scott McMillan<sup>§</sup>, José Moreira<sup>¶</sup>, Carl Yang\*,<sup>†</sup>

†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

GABB@IPDPS 2017

#### **GraphBLAS Implementations**

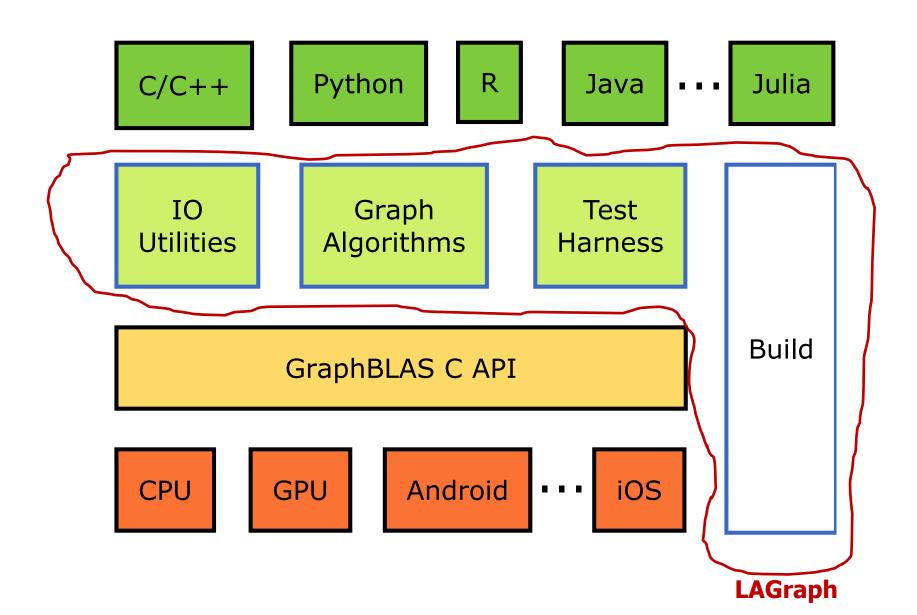
- SuiteSparse library (Texas A&M): Frist fully conforming GraphBLAS release.
  - http://faculty.cse.tamu.edu/davis/suitesparse.html
- GraphBLAS C (IBM): the second fully conforming release,
  - https://github.com/IBM/ibmgraphblas
- GBTL: GraphBLAS Template Library (CMU/SEI): Pushing GraphBLAS into C++
  - https://github.com/cmu-sei/gbtl
- GraphBLAST: A C++ implementations for GraphBLAS for GPUs (UC Davis),
  - https://github.com/gunrock/graphblast
- Python bindings:
  - PyGraphBLAS: A Python Wrapper around SuiteSparse GraphBLAS
    - https://github.com/michelp
  - PyGB: A Python Wrapper around GBTL (UW/PNNL/CMU)
    - https://github.com/jessecoleman/gbtl-python-binding
- pggraphblas: A PostgreSQL wrapper around Suite Sparse GraphBLAS
  - https://github.com/michelp
- Matlab and Julia wrappers around SuiteSparse GraphBLAS
  - https://aldenmath.com

#### **GraphBLAS Implementations**

- SuiteSparse library (Texas A&M): Frist fully conforming GraphBLAS release.
  - http://faculty.cse.tamu.edu/davis/suitesparse.html
- GraphBLAS C (IBM): the second fully conforming release,
  - https://github.com/IBM/ibmgraphblas
- GBTL: GraphBLAS Template Library (CMU/SEI): Pushing GraphBLAS into C++
  - https://github.com/cmu-sei/gbtl
- GraphBLAST: A C++ implementations for GraphBLAS for GPUs (UC Davis),
  - https://github.com/gunrock/graphblast



## The GraphBLAS Vision



## LAGraph: A curated collection of high level Graph Algorithms

Graph Algorithms built on top of the GraphBLAS.

## LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS

Tim Mattson<sup>‡</sup>, Timothy A. Davis<sup>⋄</sup>, Manoj Kumar<sup>¶</sup>, Aydın Buluç<sup>†</sup>, Scott McMillan<sup>§</sup>, José Moreira<sup>¶</sup>, Carl Yang<sup>\*,†</sup>

<sup>‡</sup>Intel Corporation <sup>†</sup>Computational Research Division, Lawrence Berkeley National Laboratory

<sup>o</sup>Texas A&M University ¶IBM Corporation §Software Engineering Institute, Carnegie Mellon University

\*Electrical and Computer Engineering Department, University of California, Davis

**GrAPL 2019** 

## The LAGraph project's official launch was at GrAPL'2019 in May'2019

## **Exercise 1: Build an LAGraph program**

- Clone our git repository
- Includes the following components
  - Exercises and solutions
  - SuiteSparse library, binaries for Linux and OSX
  - LAGraph library, binaries for Linux and OSX
- 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
$ git checkout hpec19
$ make AnalyzeGraph.exe
$ ./AnalyzeGraph.exe Data/hpec_coauthors.mtx
```

Graph of coauthors: vertices are authors, edges connect authors who've published HPEC papers together

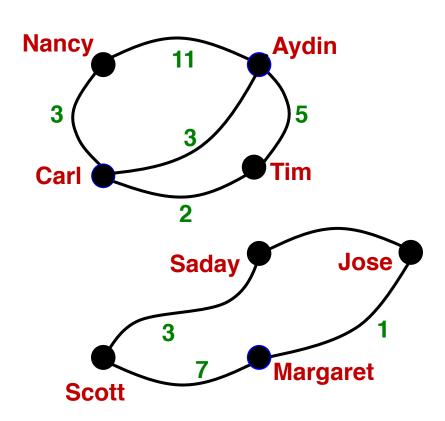
#### **Solution to Exercise 1:**

If all goes well, output should look like this:

```
$ ./AnalyzeGraph.exe Data/hpec coauthors.mtx
*** Step 1: loading input graph: Data/hpec coauthors.mtx
*** Step 1: Elapsed time: 0.00531851 sec
*** Step 2: compute some basic statistics
*** Step 2: Elapsed time: 0.00181371 sec
Num nodes: 1747
Num edges: 10072
Avg degree: 5.765312
Max degree: 461
Min degree: 1
Node with max degree (target ID): 800
*** Step 3: Running LAGraph's connected components (LACC) algorithm.
*** Step 3: Elapsed time: 0.00678869 sec
Number of connected components: 246
ID for component containing target ID 800: 0
*** Step 4: Find all the nodes from the target ID's cluster.
*** Step 4: Elapsed time: 0.000114254 sec
Cluster mask nvals (after masking): 822
Component size: 822
*** Step 5: extract and perform PageRank on the target component.
*** Step 5: Elapsed time: 0.0198434 sec
Author with the highest rank: 800 (0.019374)
Author with the smallest rank: 1094 (0.000214)
. . .
```

#### **HPEC Authors Dataset and Connected Components**

- Data represents all pairs of HPEC authors that have coauthored papers.
   The edge value represents how many papers the pair have coauthored.
- File: src/Data/hpec\_authors.mtx
- Graph (undirected):
  - 1,747 vertices (unique authors)
  - 10,072 edges (coauthor count)
- Data directory contains index tables containing the mapping between vertex ID and author name, the raw publication data, and python scripts to perform various queries



 Connected components: a subgraph of vertices connected to each other by a path, but not to vertices outside the subgraph

#### **Outline**

Graphs and Linear Algebra



- The GraphBLAS C API and Adjacency Matrices
  - GraphBLAS Operations
  - Breadth-First Traversal
  - Connected Components

## **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**

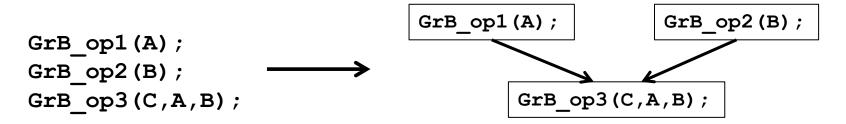
- Opaque object: An object manipulated strictly through the GraphBLAS API whose implementation is not defined by the GraphBLAS specification.
- Transparent object: an object whose structure is fully exposed to the programmer. E.g.: an array of tuples <i, j, value>
- Method: Any C function that manipulates 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. <a href="http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf">http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf</a>
  - Examples: matrix multiply, matrix-vector multiply, reduction, apply

## **Code from our first example (EXERPTS)**

```
#include <stdio.h>
#include "GraphBLAS.h"
#include "LAGraph.h"
#include "tutorial utils.h"
int main(int argc, char** argv) {
   LAGraph init();
                                                  Initialize a LAGraph and GraphBLAS
   FILE *fd = fopen(argv[1], "r");
   GrB Matrix graph = NULL;
                                                       Create and load a matrix object
   if (GrB SUCCESS != LAGraph mmread(&graph, fd))
                                                       with data from a file.
      exit(-1);
   GrB Index nrows, ncols, nvals;
   GrB Matrix nrows(&nrows, graph);
                                                  Query the matrix for the dimensions
                                                  and number of defined (stored) values
   GrB Matrix ncols(&ncols, graph);
   GrB Matrix nvals(&nvals, graph);
   printf("Num nodes: %ld\n", nrows);
   printf("Num edges: %ld\n", nvals);
                                                  Find the connected components
   LAGraph lacc(A, &components);
   // Cleanup
                                                  Free memory used for our matrix
   GrB free(&A);
   LAGraph finalize();
                                                  Close the context, release resources
}
```

#### **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.)



- 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	{false, 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 <sub>-</sub> FP32	float	IEEE 754 binary32
GrB_FP64	double	IEEE 754 binary64

## Exercise 2: Build a GraphBLAS program

- Build your first GraphBLAS program
- Build and run our BuildGraph.exe program

```
$ cd GraphBLAS/src
$ make BuildGraph.exe
$ ./BuildGraph.exe
```

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

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

### Code from our BuildGraph exercise

```
#include <stdio.h>
#include <assert.h>
#include <GraphBLAS.h>
#include "tutorial utils.h"
int main(int argc, char** argv) {
                                                Initialize a context in BLOCKING mode
   GrB init(GrB BLOCKING);
                                                GrB Index used for matrix dimension
   GrB Index const NUM NODES = 3;
   GrB Matrix graph;
                                                Create a matrix object of order
   GrB Matrix new(&graph, GrB UINT64,
                                                NUM NODES and domain UINT64
                    NUM NODES, NUM NODES);
                                                Store the value 4 in element (1,2)
   GrB Matrix setElement(graph, 4, 1, 2);
                                                     Our own "pretty print" routine
   pretty_print_matrix_UINT64(graph, "GRAPH");
                                                      (not part of GraphBLAS)
   GrB Index nvals;
                                                Query the matrix for the number of
   GrB Matrix nvals(&nvals, graph);
                                                defined (stored) values and check for
   assert(nvals == 1);
                                                correctness
   // Cleanup
                                                Free memory used for our matrix
   GrB free(&graph);
   GrB finalize();
                                                Close the context, release resources
```

26

## **Exercise 3: 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

- Hint: Save time and minimize typing
  - Copy BuildGraph.c into another file 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 3: 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.

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

manipulated but the handle doesn't change,

we pass by value (i.e. without the &).

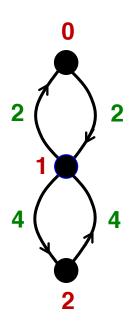
- Output the result and verify that your adjacency graph is correct.
- You will need the following types and methods from the GraphBLAS

- Hint: Save time and minimize typing
  - Copy BuildGraph.c into another file 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.

#### **Solution to Exercise 3**

```
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

- 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,
```

#### 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_ <i>T</i>	$T \times T \rightarrow bool$	f(x,y) = (x==y)	Equal
GrB_MIN_ <i>T</i>	$T \times T \rightarrow T$	f(x,y) = (x < y)?x:y	minimum
GrB_MAX_T	$T \times T \rightarrow T$	f(x,y) = (x>y)?x:y	maximum
GrB_PLUS_T	$T \times T \rightarrow T$	f(x,y) = x + y	addition
GrB_TIMES_T	$T \times T \rightarrow T$	f(x,y) = x * y	multiplication
GrB_FIRST_T	$T \times T \rightarrow T$	f(x,y) = x	First argument
GrB_SECOND_T	$T \times T \rightarrow T$	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

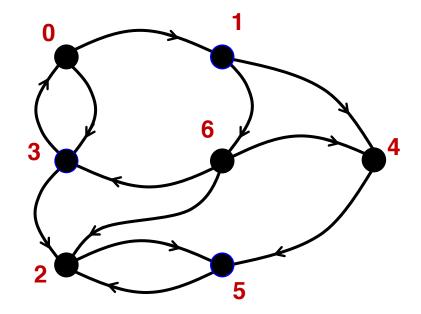
This is a subset of the defined types and operators. See table 2.3 for the full list.

## 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 4: Adjacency matrix**

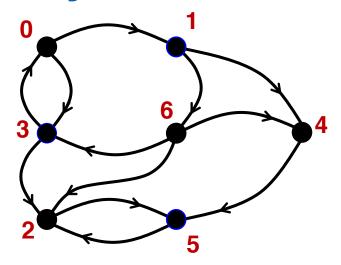
 Write a program to create the adjacency matrix for the GraphBLAS "logo" graph using row, column and value arrays.



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 4



GrB Index const NUM NODES = 7;

```
Matrix: Graph =

[ -, 1, -, 1, -, -, -]
[ -, -, -, -, 1, -, 1]
[ -, -, 1, -, -, -, 1, -]
[ 1, -, 1, -, -, -, -, -]
[ -, -, 1, -, -, -, -]
[ -, -, 1, 1, 1, -, -]
```

#### **Outline**

- Graphs and Linear Algebra
- The GraphBLAS C API and Adjacency Matrices



- GraphBLAS Operations
  - Breadth-First Traversal
  - Connected Components

#### **GraphBLAS Operations** (from the Math Spec\*)

Operation name	Mathematical description					
mxm	$\mathbf{C}$	<u>⊙</u> =	${f A} \oplus . \otimes {f B}$			
mxv	$\mathbf{w}$		$\mathbf{A} \oplus . \otimes \mathbf{v}$			
vxm	$\mathbf{w}^T$	$\odot =$	$\mathbf{v}^T \oplus . \otimes \mathbf{A}$			
eWiseMult	${f C}$	$\odot$ =	$\mathbf{A} \otimes \mathbf{B}$			
	$\mathbf{w}$	$\odot =$	$\mathbf{u} \otimes \mathbf{v}$			
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			$F_u(\mathbf{A})$			
	$\mathbf{w}$	$\odot =$	$F_{u}(\mathbf{u})$			
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.

<sup>\*</sup> Mathematical foundations of the GraphBLAS, Kepner et. al. HPEC'2016

$$\mathsf{w} \odot = \mathsf{A} \oplus . \otimes \mathsf{u}$$

Multiply a matrix times a vector to produce a vector

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

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

#### **Definitions:**

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

$$\mathsf{w} \odot = \mathsf{A} \oplus . \otimes \mathsf{u}$$

Multiply a matrix times a vector to produce a vector

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

The summation is over the intersection of the existing elements in the i<sup>th</sup> 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

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

#### **Definitions:**

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

$$\mathsf{w} \odot = \mathsf{A} \oplus . \otimes \mathsf{u}$$

- 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()



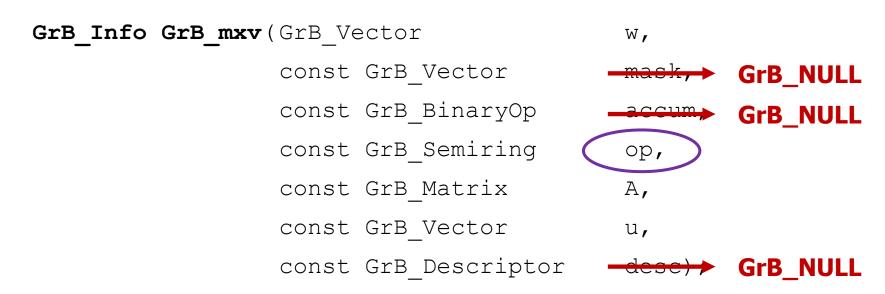
- Compute the product of a GraphBLAS sparse matrix with a GraphBLAS vector.
- Returns error codes of type GrB\_info. See the spec for details.

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

## GrB\_mxv()



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



Op defines the algebraic structure, a semiring in this case. This gives us  $\otimes$  and  $\oplus$  and the identity for  $\oplus$ . 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 5: Matrix Vector Multiplication**

- Use the adjacency matrix from exercise 4 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 4, you'll need the functions:

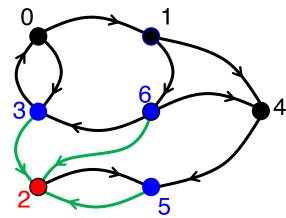
```
3
```

#### Solution to exercise 5

```
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");
```

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



```
Matrix: GRAPH =

[ -, 1, -, 1, -, -, -]
[ -, -, -, -, 1, -, 1]
[ -, -, -, -, -, 1, -]
[ 1, -, 1, -, -, -, -]
[ -, -, -, -, -, 1, -]
[ -, -, 1, 1, 1, -, -]
Vector: Target node =

[ -, -, 1, -, -, -, -]
Vector: sources =

[ -, -, -, 1, -, 1, 1]
```

## 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)

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_INPO: 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 6: Matrix Vector Multiplication**

- Modify your program from exercise 5 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 INPO, GrB INP1, GrB OUTP, GrB MASK
                  VALUE: GrB TRAN, GrB REPLACE, GrB SCMP
```

#### Solution to exercise 6

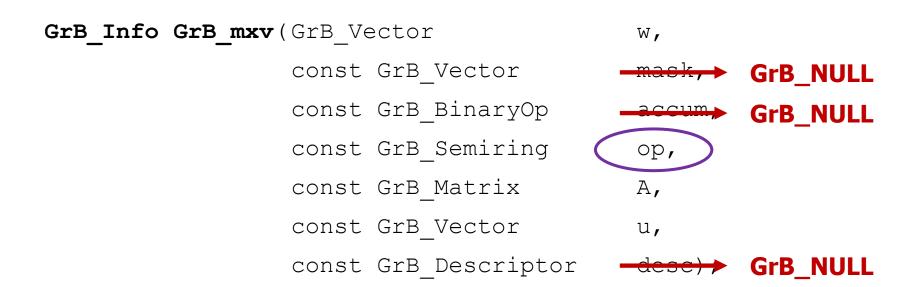
```
// Build a vector with one node set.
GrB Index const SRC NODE = 6;
GrB Vector vec;
GrB Vector new(&vec, GrB BOOL, NUM NODES);
GrB Vector setElement(vec, true, SRC NODE);
                                                Vector: source node =
GrB Descriptor desc;
GrB Descriptor new(&desc);
                                                Vector: neighbors =
                                                [-, -, 1, 1, 1, -, -]
GrB_Descriptor_set(desc, GrB_INP0, GrB_TRAN);
                                                GrB mxv test passed.
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");
```

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

## GrB\_mxv()



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



It's time to explain semirings in GraphBLAS operations

#### **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	)			Standa	rd ope	erations in linear algebra
Notation:	(R,	+,	*,	0,	1)	
	Scalar type	Op1	Op2	Identity Op1	Identity Op2	

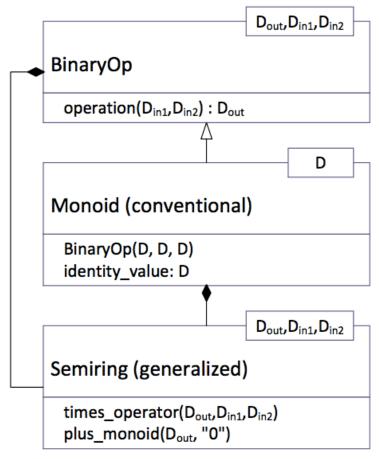
## **Algebraic Semirings**

- 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:

$$M = \langle D, \bigoplus, 0 \rangle$$

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

$$S = \langle D_{out}, D_{in1}, D_{in2}, M, \otimes \rangle$$

The domains must be consistent:

$$\bigotimes: D_{in1} \times D_{in2} \to D_{out}$$
 $\bigoplus: D_{out} \times D_{out} \to D_{out}$ 
 $0 \in D_{out}$ 

## **Building Semirings in the GraphBLAS**

First you build the monoid (M) for the "addition" and its 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"):

• 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	-INFINITY	GrB_SECOND_FP32

#### **Exercise 7: 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, Oul);
- GrB_Semiring UInt64Arith;
- GrB_Semiring_new(&UInt64Arith, UInt64Plus, GrB_TIMES UINT64);
```

# To save some time, we will skip this exercise

#### **Outline**

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



- Breadth-First Traversal
  - Connected Components

#### **Breadth First Traversal**

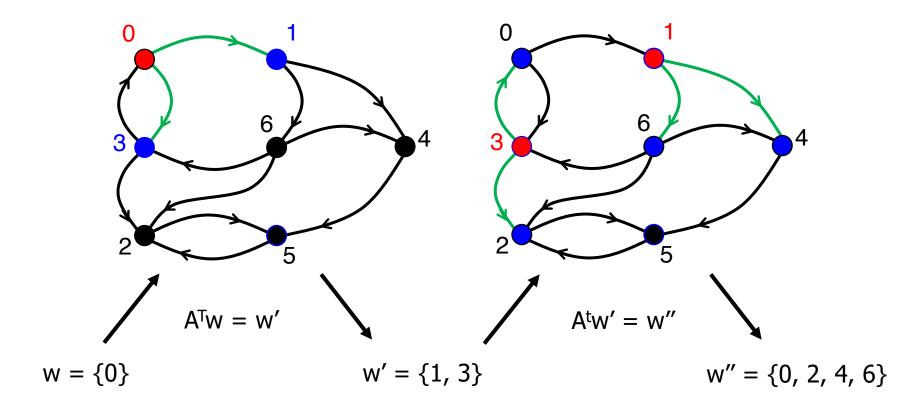
- The Breadth First Traversal:
  - Start from one or more initial vertices
  - Visit all accessible one hop neighbors,
  - Visit all accessible unique two hop neighbors,
  - Continue until no more unique vertices to visit
  - Note: keep track of vertices visited so you don't visit the same vertex more 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 (see HPEC18)
  - Construct a Connected Components 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

A = Adjacency Matrix

w = wavefront vector

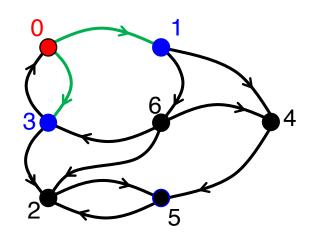
#### **Exercise 8: Traverse the graph**

- Modify your code from Exercises 6 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 8

```
// 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_INPO, 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");
```

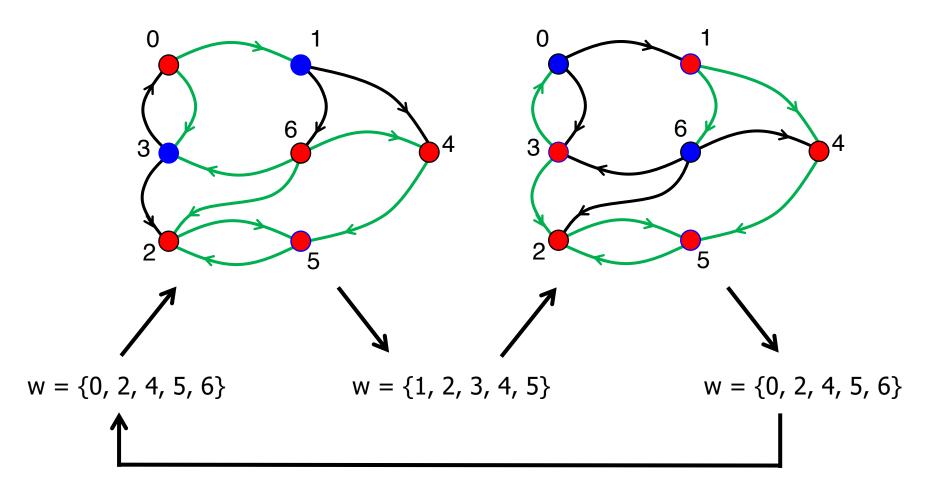


```
Vector: wavefront(src) =
Γ 1, -, -, -,
Vector: wavefront =
[-, 1, -, 1,
Vector: wavefront =
[ 1, -, 1, -, 1, -, 1]
Vector: wavefront =
                 1, 1, -]
[ -, 1, 1, 1,
Vector: wavefront =
                 1, 1, 1]
      -, 1, -,
Vector: wavefront =
[ -, 1, 1, 1, 1, -]
Vector: wavefront =
[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

• "We tell a bunch, and they tell bunch...(rinse and repeat)"

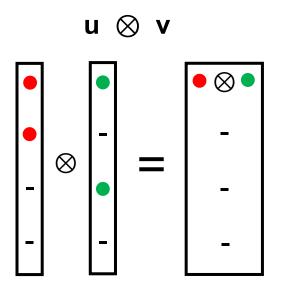


#### **Visited lists**

- Breadth-first traversal requires that we only need to 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.

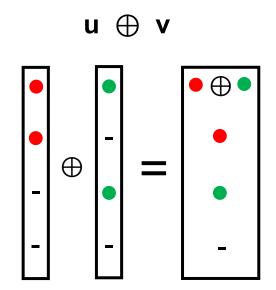
#### **Element-wise Operations: Mult and Add**

 ⊗ assumes unstored values (-) are the binary operator's annihilator:



Examples: (x,0), (and, false),  $(+, \infty)$ 

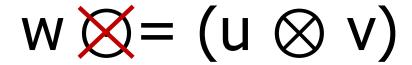
 ⊕ assumes unstored values (-) are the binary operator's identity:



Examples: (+,0), (or, false), (min,  $\infty$ )

The rules for element-wise addition also apply to the accumulation operator, ⊙

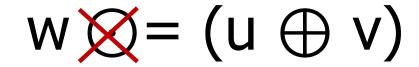
## GrB\_eWiseMult()



- 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.

Use default values for mask, accum and desc (indicated by GrB\_NULL)

## GrB\_eWiseAdd()



- 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.

Use default values for mask, accum and desc (indicated by GrB\_NULL)

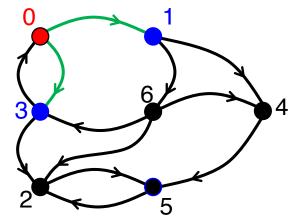
#### Exercise 9: Keep track of 'visited' nodes

 Modify code from Exercise 8 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 9

```
// 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 INPO, GrB TRAN);
pretty_print_vector_UINT64(w, "wavefront(src)"); [ 1, 1, 1, -, 1, -, -]
for (int i=0; i<NUM NODES; ++i) {</pre>
  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");
```

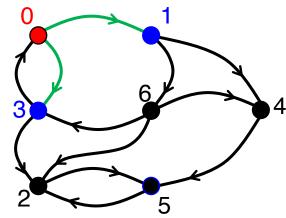


```
Vector: wavefront(src) =
Vector: visited =
[ 1, -, -, -, -, -]
Vector: wavefront =
[ -, 1, -, 1, -, -, -]
Vector: visited =
Vector: wavefront =
[ 1, -, 1, -, 1, -,
Vector: visited =
[1, 1, 1, 1, 1, -, 1]
Vector: wavefront =
[ -, 1, 1, 1, 1, -]
Vector: visited =
[ 1, 1, 1, 1, 1, 1]
Vector: wavefront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

[ 1, 1, 1, 1, 1,

#### Solution to exercise 9

```
// 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(desq
                       What should the
pretty_print_vector_undering exit condition be?
for (int i=0; i<NUM NODES; ++i) {</pre>
  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");
                                                 [ 1, 1, 1, 1, 1,
```



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

GrB\_mxv() 
$$w(\neg m, z) \boxtimes = (A \oplus . \otimes u)$$

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

```
GrB_Info GrB_mxv(GrB_Vector w,

const GrB_Vector mask,

const GrB_BinaryOp accum, GrB_NULL

const GrB_Semiring op,

const GrB_Matrix A,

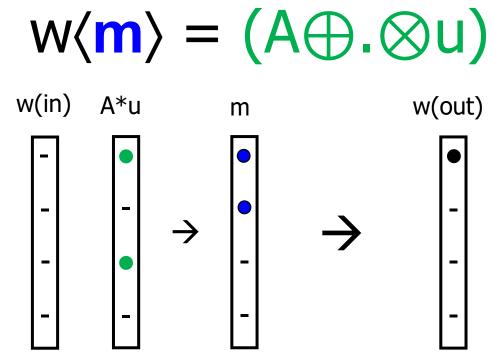
const GrB_Vector u,

const GrB_Descriptor desc);
```

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

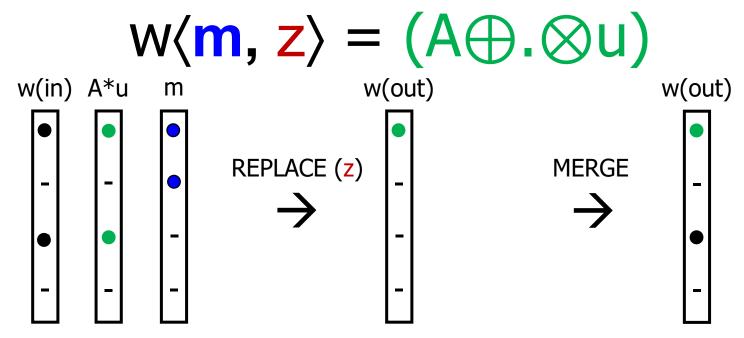
# Masking

- 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



#### REPLACE vs. "MERGE"

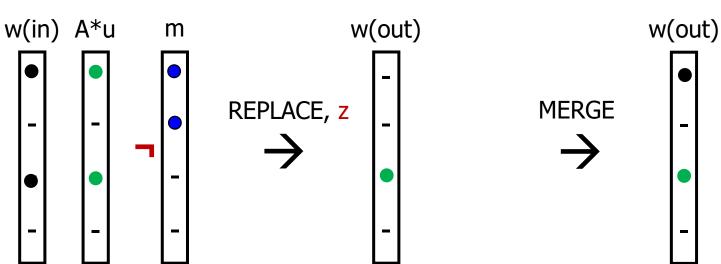
- When a mask is used and the output container is not empty when the operation is called...what do you do to the "masked out" elements?
  - REPLACE (z): all unwritten locations are cleared (zeroed out).
  - MERGE: all unwritten locations are left alone.
- Behaviour defaults to MERGE; otherwise, use a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



#### 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 INPO: Grb TRAN (transpose)
  - Grb INP1: Grb TRAN (transpose)
  - Grb\_MASK: Grb\_SCMP (structural complement, Grb\_COMP in v1.3)
  - Grb OUTP: Grb REPLACE (clear the output before writing result)

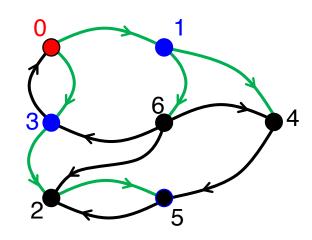
#### **Exercise 10: Avoid revisiting**

- 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 Vector nvals(&nvals vec);
- GrB eWiseAdd(vec, GrB NULL, GrB NULL,
               GrB_LOR, vec, wav, GrB_NULL);
- GrB mxv(result, mask, GrB NULL,
          GxB LOR LAND BOOL, graph, vec, desc);
- GrB Descriptor desc;
- GrB Descriptor new(&desc);
- GrB Descriptor set(desc, FIELD, VALUE)
  FIELD: GrB INPO, GrB INP1, GrB OUTP, GrB MASK
  VALUE: GrB_TRAN,
                 GrB REPLACE, GrB SCMP
```

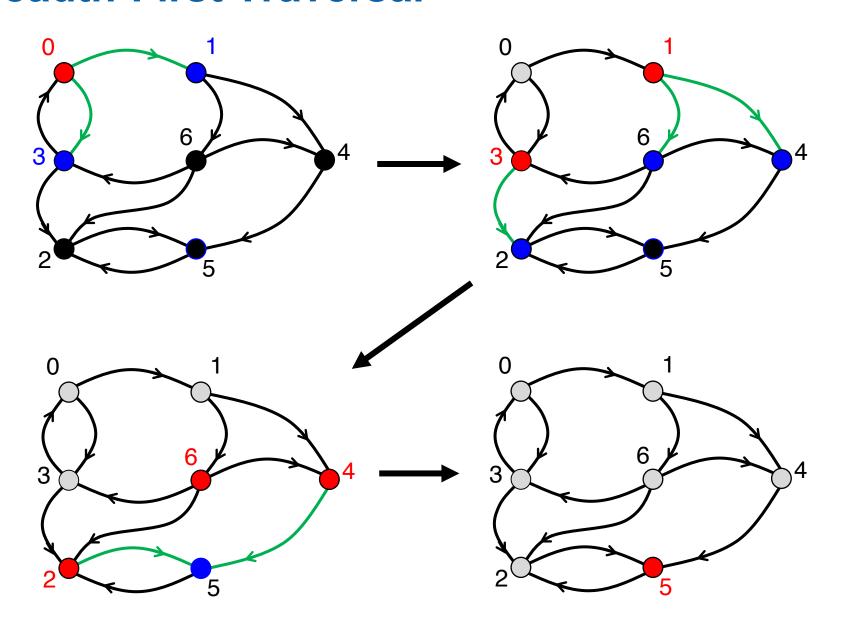
#### Solution to exercise 10

```
GrB Vector setElement(w, true, SRC NODE);
GrB Descriptor desc;
GrB Descriptor new(&desc);
GrB Descriptor set(desc, GrB INPO, 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,
Vector: wavefront =
                         81
```

#### **Breadth-First Traversal**



#### **Outline**

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



Connected Components

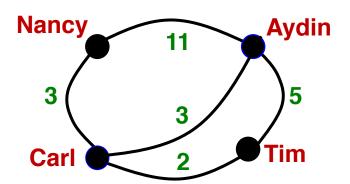
#### **Connected Components**

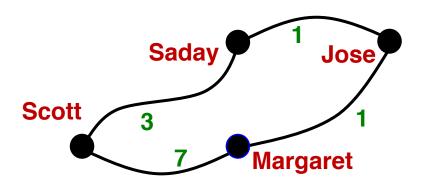
#### Connected Components

- Identify groups of vertices with paths to one another.
- Identify how many of these groups (components) exist in the data.
- Goal: assign all vertices within a component with the same unique
   ID.

#### Graph

- Consists of undirected edges
- Note: applying this to directed graphs by converting to undirected is called "weakly connected components."





# GrB\_assign()

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

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

- Assign a vector to the elements of column c<sub>i</sub> of a matrix
- Assign a vector to the elements of row r<sub>i</sub> of a matrix

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

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

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

# GrB\_assign() from vector $w(i) \odot = u$

$$\mathbf{w}(i) \odot = \mathbf{u}$$

- 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),
                       \forall j : 0 \le j < \text{nindices},
  w(indices[j]) = w(indices[j]) \bigcirc u(j), \forall j : 0 \le j < nindices.
GrB Info GrB_assign( GrB_Vector
                                                   W,
                      const GrB Vector
                                                   mask,
                      const GrB BinaryOp
                                                   accum,
                      const GrB Vector
                                                  u,
                                                  *indices,
                      const GrB Index
                      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.

# **GrB\_assign() from constant** $\mathbf{w}(i) \odot = c$

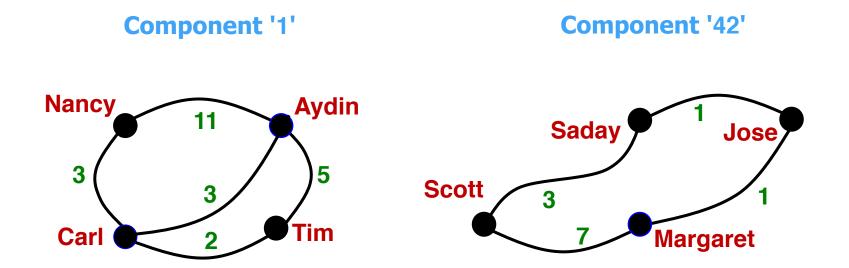
- Assign a constant to a subset of the output vector.
- Locations to be assigned selected by an output index vector, indices:

```
\forall j : 0 \le j < \text{nindices},
  w(indices[i]) = c
  w(indices[j]) = w(indices[j]) \odot c, \forall j : 0 \le j < nindices.
GrB Info GrB assign (GrB Vector
                                                    W,
                      const GrB Vector
                                                    mask,
                      const GrB BinaryOp
                                                    accum,
                      <type>
                                                    C,
                                                   *indices,
                      const GrB Index
                      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).

#### **Our Connected Components plan**

- Strategy:
  - Initialize all vertex IDs to "unassigned"
  - While there are unassigned vertices:
    - Pick an unassigned vertex
    - Perform BFS marking all reachable vertices
    - Assign all reachable vertices with a unique 'component number'.



#### **Our Connected Components plan**

 We need an undirected graph with disconnected components to play with:

## Our Connected Components plan

 We need an undirected graph with disconnected components to play with:

row ind: {0, 0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 8, 8};

```
col ind: {1, 4, 8, 0, 8, 6, 5, 7, 0, 8, 3, 7, 2, 3, 5, 0, 1, 4};
values: {1, 2, 1, 1, 3, 4, 1, 2, 2, 5, 1, 2, 4, 2, 2, 1, 3, 5};
Matrix: GRAPH =
[ -, 1, -, -, 2, -,
                               3]
                               -]
[ -, -, -, -, 1, -, 2, -]
[ 2, -, -, -, -, 5]
[ -, -, -, 1, -, -, -, 2, -]
[ -, -, 4, -, -, -, -,
                               -]
[ -, -, -, 2, -, 2, -,
                               -]
     3, -, -, 5, -, -, -,
                               -]
```

How many components are there?

#### **Exercise 11: Connected Components**

Wrap the code from Exercise 10 in a function:

```
- GrB_Info BFS(GrB_Matrix const graph,
GrB_Index src_node,
GrB_Vector visited); // return GrB_SUCCESS
```

- Call this function to compute the membership of each connected component (CC):
  - Create a vector of size NUM NODES to hold CC ID for each node.
  - Each CC consists of all reachable (visited) nodes from a given root.
  - Use the following undirected, weighted graph, with multiple components:

```
row_ind: {0, 0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 8, 8};
col_ind: {1, 4, 8, 0, 8, 6, 5, 7, 0, 8, 3, 7, 2, 3, 5, 0, 1, 4};
values: {1, 2, 1, 1, 3, 4, 1, 2, 2, 5, 1, 2, 4, 2, 2, 1, 3, 5}; // pub. count
```

Challenge: use GrB\_assign to assign component IDs

```
- GrB_Vector_new(&vec, GrB_UINT64, NUM_NODES);
- GrB_Vector_setElement(w, true, SRC_NODE);
- GrB_Vector_extractElement(&s, w, index);
- pretty_print_vector_UINT64(vec, "CC IDS");
- GrB_assign(u, mask, accum, s, GrB_ALL, NUM_NODES, desc);
- GrB_mxv(w, mask, accum, GxB_LOR_LAND_BOOL, graph, w, desc);
- GrB_Vector_nvals(&nvals, w);
```

# Solution to exercise 11 (part1)

```
5
GrB Info BFS (GrB Matrix const graph,
             GrB Index
                              src node,
             GrB Vector
                              v)
  GrB Index num nodes;
  GrB Matrix nrows(&num nodes, graph);
  GrB Vector w;
  GrB Vector new(&w, GrB BOOL, NUM NODES);
  GrB Vector setElement(w, true, src node);
  GrB Descriptor desc;
  GrB Index nvals = 0;
  do {
    //GrB eWiseAdd(v, GrB NULL, GrB NULL, GrB LOR, v, w, GrB NULL);
    GrB assign(v, w, GrB NULL, true, GrB ALL, num nodes, GrB NULL);
    GrB mxv(w, v, GrB NULL, GxB LOR LAND BOOL, graph, w, desc);
    GrB Vector nvals(&nvals, w);
  } while (nvals > 0);
  return GrB SUCCESS;
```

## Solution to exercise 11 (part2)

```
GrB Index tmp = 0, num ccs = 0;
GrB Vector cc ids, visited;
GrB Vector new(&cc ids, GrB UINT64, NUM NODES);
GrB Vector new(&visited, GrB BOOL, NUM NODES);
for (GrB Index src = 0; src < NUM NODES; ++src) {</pre>
  // find next unassigned node
  if (GrB NO VALUE == GrB VectorExtractElement(
                                 &tmp, cc ids, src)) {
    BFS(graph, src, visited);
    // cc ids[visited] = src
    GrB assign(cc ids, visited, GrB NULL,
               src, GrB ALL, NUM NODES, GrB NULL);
  GrB Vector clear(visited);
printf("Number of connected components: %ld\n", num ccs);
pretty print vector UINT64(components, "CC IDs");
```

Output:

```
Number of connected components: 3

Vector: CC IDs =

[ 0, 0, 2, 3, 0, 3, 2, 3, 0]
```

5

## Putting it all together...

Comparing with LAGraph's CC algorithm\* on HPEC dataset

```
$ Solutions/AnalyzeGraph final.exe ../Data/hpec coauthors.mtx
*** Step 3: Running Tutorial connected components algorithm.
Largest component #0 (size = 822)
*** Step 3: Elapsed time: 0.0189476 sec
Number of connected components: 246
ID for component containing target ID 800: 0
*** Step 3: Running LAGraph's connected components (LACC) algorithm.
*** Step 3: Elapsed time: 0.0066211 sec
Number of connected components: 246
ID for component containing target ID 800: 0
```

Platform: Dell Precision 7510, Intel Xeon CPU E3-1505M v5, 64GB RAM, VMWare 14.1.7, Ubuntu 16.0.4LTS

#### The GraphBLAS Operations

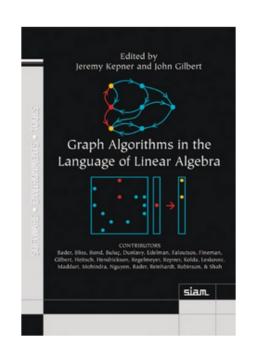
Operation Name	Matl	nema	atical No	tati	ion
mxm	$\mathbf{C}\langle\mathbf{M},z angle$	=	C	0	$\mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m},z angle$	=	$\mathbf{w}$	$\odot$	$\mathbf{A} \oplus . \otimes \mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T,z angle$	=	$\mathbf{w}^T$	$\odot$	$\mathbf{u}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C}\langle\mathbf{M},z angle$	=	$\mathbf{C}$	$\odot$	$\mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m},z angle$	=	$\mathbf{w}$	$\odot$	$\mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C}\langle\mathbf{M},z angle$	=	$\mathbf{C}$	$\odot$	$\mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w}\langle\mathbf{m},z\rangle$	=	$\mathbf{w}$	$\odot$	$\mathbf{u}\oplus\mathbf{v}$
reduce (row)	$\mathbf{w}\langle\mathbf{m},z angle$	=	$\mathbf{w}$	$\odot$	$[\oplus_j \mathbf{A}(:,j)]$
reduce (scalar)	s	=	s	$\odot$	$[\oplus_{i,j} \mathbf{A}(i,j)]$
	s	=	s	$\odot$	$[\oplus_i \mathbf{u}(i)]$
apply	$\mathbf{C}\langle\mathbf{M},z angle$	=	$\mathbf{C}$	$\odot$	$f_u(\mathbf{A})$
	$\mathbf{w}\langle\mathbf{m},z angle$	=	$\mathbf{w}$	$\odot$	$f_u(\mathbf{u})$
transpose	$\mathbf{C}\langle\mathbf{M},z angle$	=	$\mathbf{C}$	$\odot$	$\mathbf{A}^T$
extract	$\mathbf{C}\langle\mathbf{M},z angle$	=	$\mathbf{C}$	$\odot$	$\mathbf{A}(m{i},m{j})$
	$\mathbf{w}\langle\mathbf{m},z\rangle$	=	$\mathbf{w}$	$\odot$	$\mathbf{u}(i)$
assign	$\mathbf{C}\langle \mathbf{M},z angle (m{i},m{j})$	=	$\mathbf{C}(m{i},m{j})$	$\odot$	A
	$\mathbf{w}\langle\mathbf{m},z angle(m{i})$	=	$\mathbf{w}(i)$	$\odot$	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 Architectures Programming and Learning (GrAPL) workshop at IPDPS
  - Attend GraphBLAS BoFs at HPEC and Supercomputing
  - Explore the challenge problems included with this tutorial
  - Work through the algorithms in the Graph book →



## **GraphBLAS at HPEC 2019**

- 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)
   Wednesday from 6 PM to 7 PM, Eden Vale C1.
- Please give us feedback about the tutorial by filling out the survey or sending email:

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.

$$C = A \oplus . \otimes B = AB$$

Matrix Multiplication ... the way we learned it in school

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

 $\mathbf{A}:\mathbb{S}^{m imes l}$ 

 $\mathbf{B}:\mathbb{S}^{l imes n}$ 

 $\mathbf{C}:\mathbb{S}^{m imes n}$ 

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
С	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.
Α	GrB_INP0	GrB_TRAN	Use transpose of A for operation.
В	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_INPO, 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 <sup>T</sup>
Test the domains and sizes for consistency.	int32, dims match
Carry out the indicated operation	T ← A *.+ B, Z ← 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 ← 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

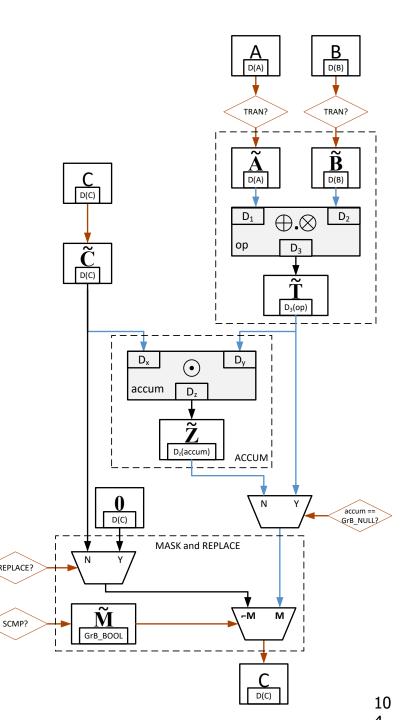
To understand what happens inside a graphBLAS operation, consider matrix multiply.

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);
```

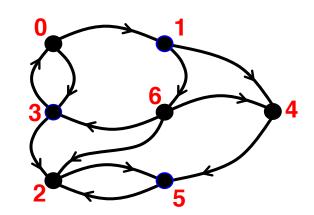
M

D(M)



#### **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<sup>T</sup>
    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 arbitrarysized 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 in-
	compatible 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 ac-
	cessed 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
GrB_INVALID_OBJECT	output. One of the opaque GraphBLAS objects (input or output) is in an invalid state caused by a
GrB_INDEX_OUT_OF_BOUNDS	previous execution error.  Reference to a vector or matrix element that is outside the defined dimensions of the object.
GrB_PANIC	Unknown internal error.