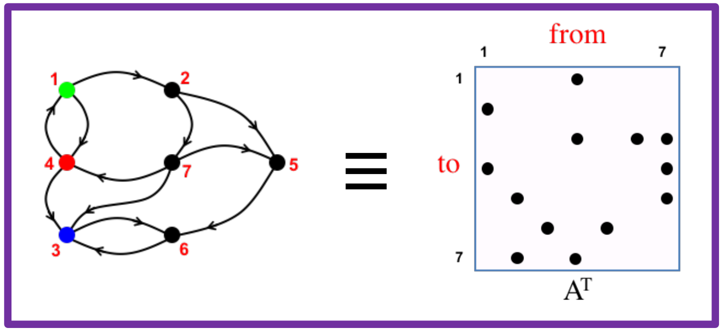


A Hands-On Introduction to GraphBLAS:
The Python Edition

http://graphblas.org

Scott McMillan
CMU/SEI

Tim Mattson
Intel Labs

… and the other members of the GraphBLAS specification group:
Aydın Buluç (UC Berkeley/LBNL), Jose Moreira (IBM), and **Ben Brock (UC Berkeley).**

With a special thank you to **Tim Davis (Texas A&M)** for GraphBLAS support in SuiteSparse and **Michel Pelletier (Graphegon)** for creating pygraphblas.

To get course materials onto your laptop:

```
$ git clone -b classroom21 https://github.com/GraphBLAS/SIAM-Tutorial.git
```

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.

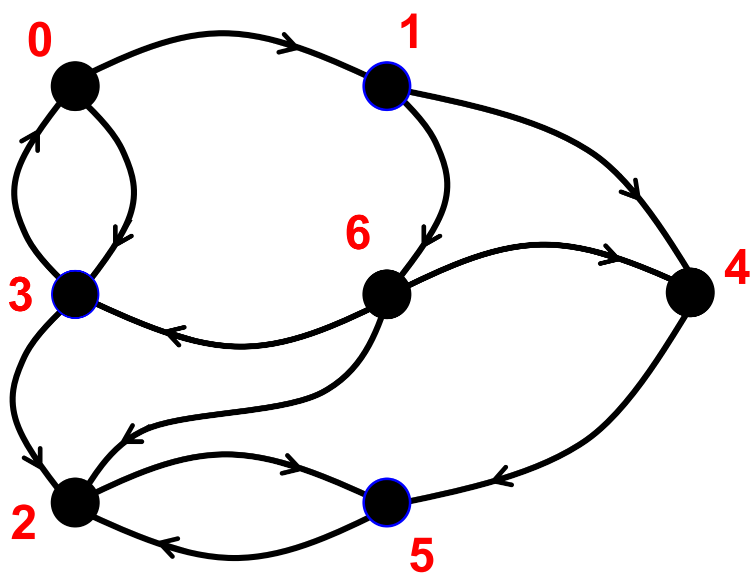
DM21-0192

Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - Breadth-First Traversal
  - Connected Components

Understanding relationships between items

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



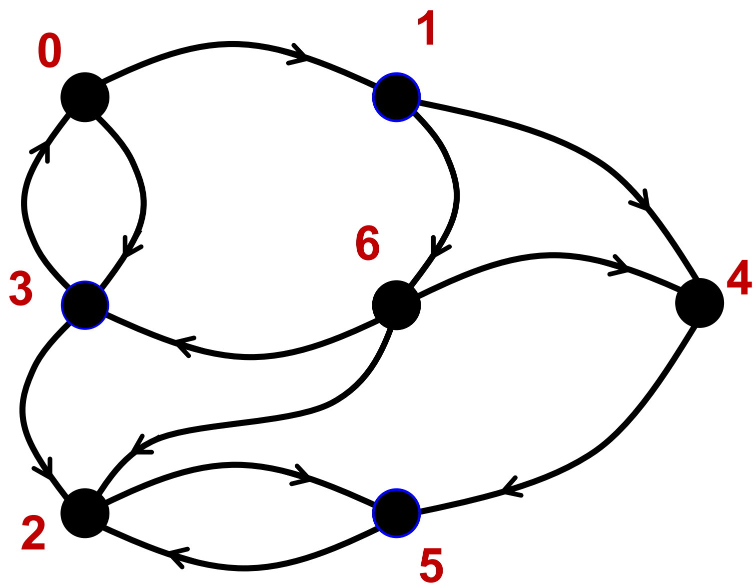
- Graph: Two sets, one for the vertices (v) and one for the edges (e)

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

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

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



To vertex (columns)

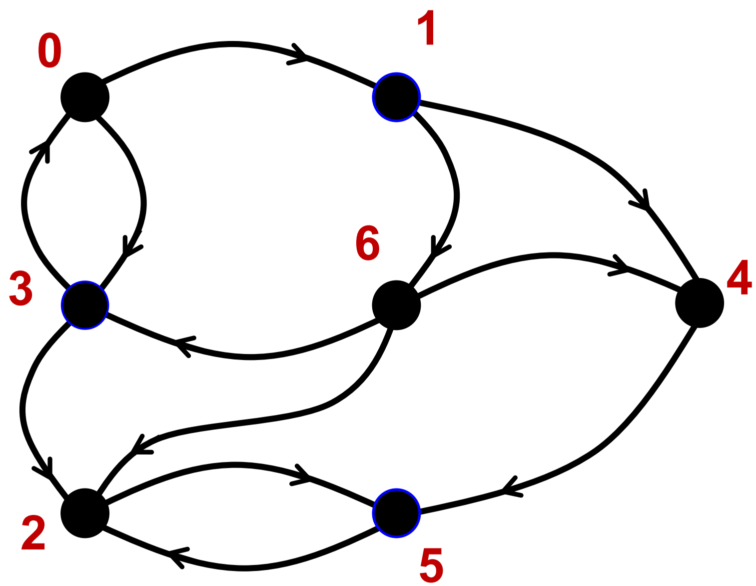
$$A = \begin{pmatrix} - & \star & - & \star & - & - & - \\ - & - & - & - & \star & - & \star \\ - & - & - & - & - & \star & - \\ \star & - & \star & - & - & - & - \\ - & - & - & - & - & \star & - \\ - & - & \star & - & - & - & - \\ - & - & \star & \star & \star & - & - \end{pmatrix}$$

From vertex (rows)

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

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



To vertex (columns)

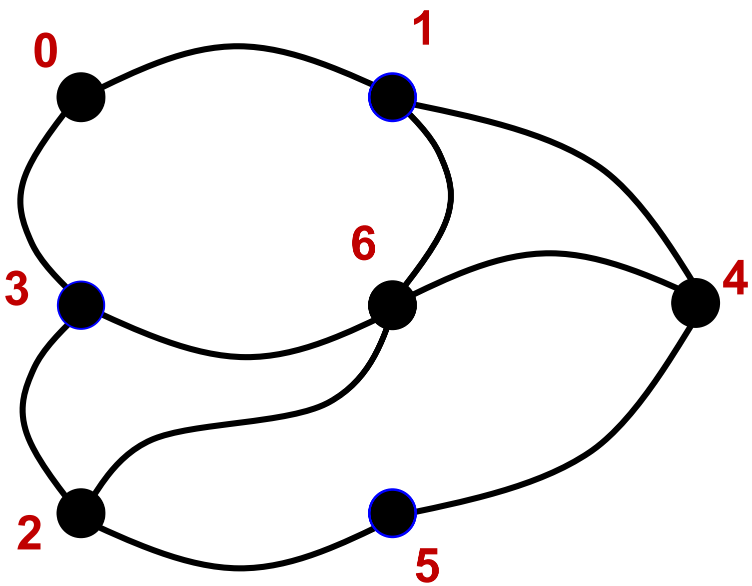
From vertex (rows)

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

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



To vertex (columns)

From vertex (rows)

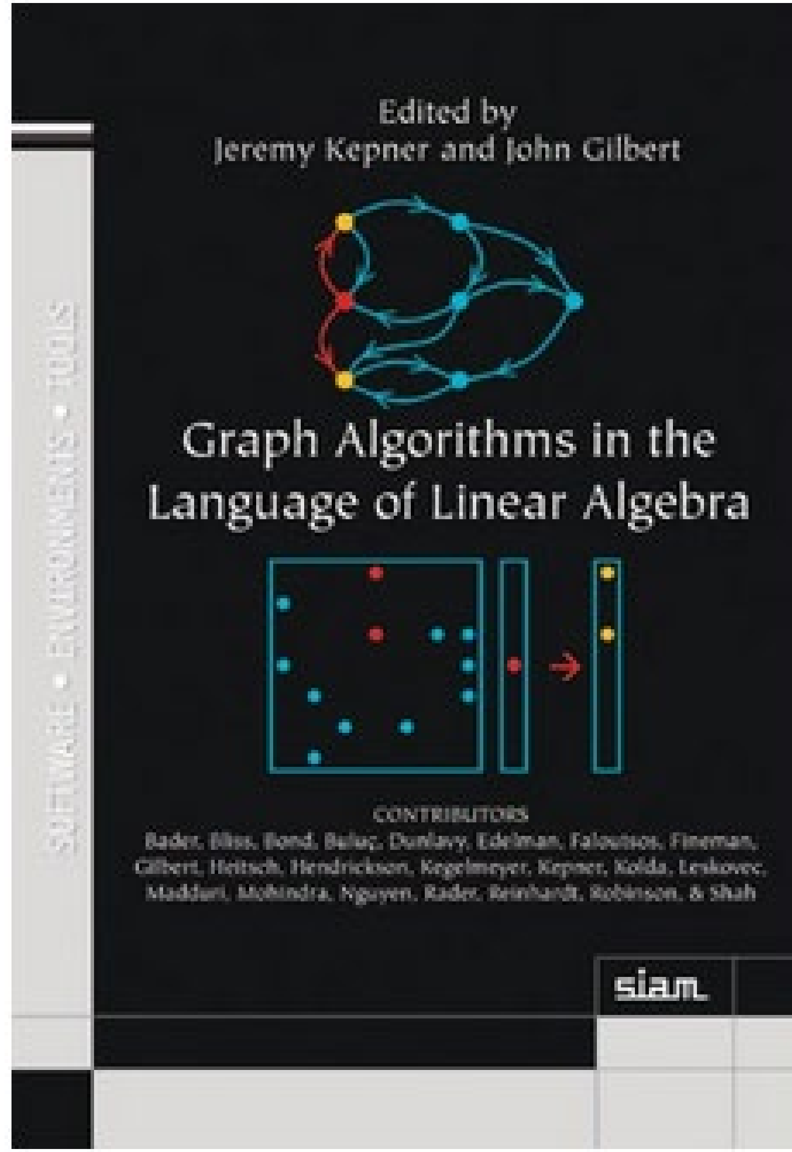
$$A = \begin{pmatrix} - & \star & - & \star & - & - & - \\ \star & - & - & - & \star & - & \star \\ - & - & - & \star & - & \star & \star \\ \star & - & \star & - & - & - & \star \\ - & \star & - & - & - & \star & \star \\ - & - & \star & - & \star & - & - \\ - & \star & \star & \star & \star & - & -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

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 **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

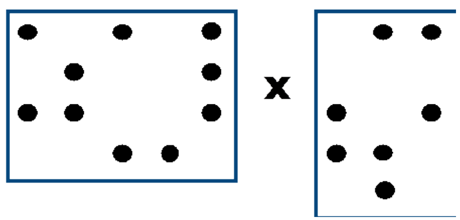
- The BLAS supports a separation of concerns:
  - HW/SW optimization experts tuned the BLAS for specific platforms.
  - Linear algebra experts build software on top of the BLAS ... high performance “for free”.

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

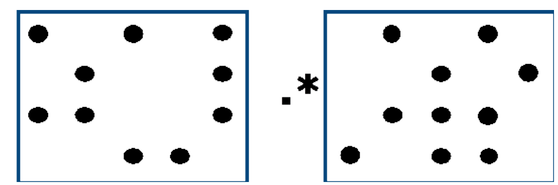
GraphBLAS: building blocks for graphs as linear algebra

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

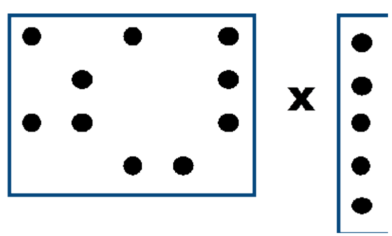
Matrix multiplication



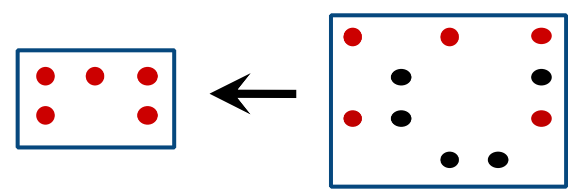
Element-wise operations (eWiseAdd, eWiseMult)



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*

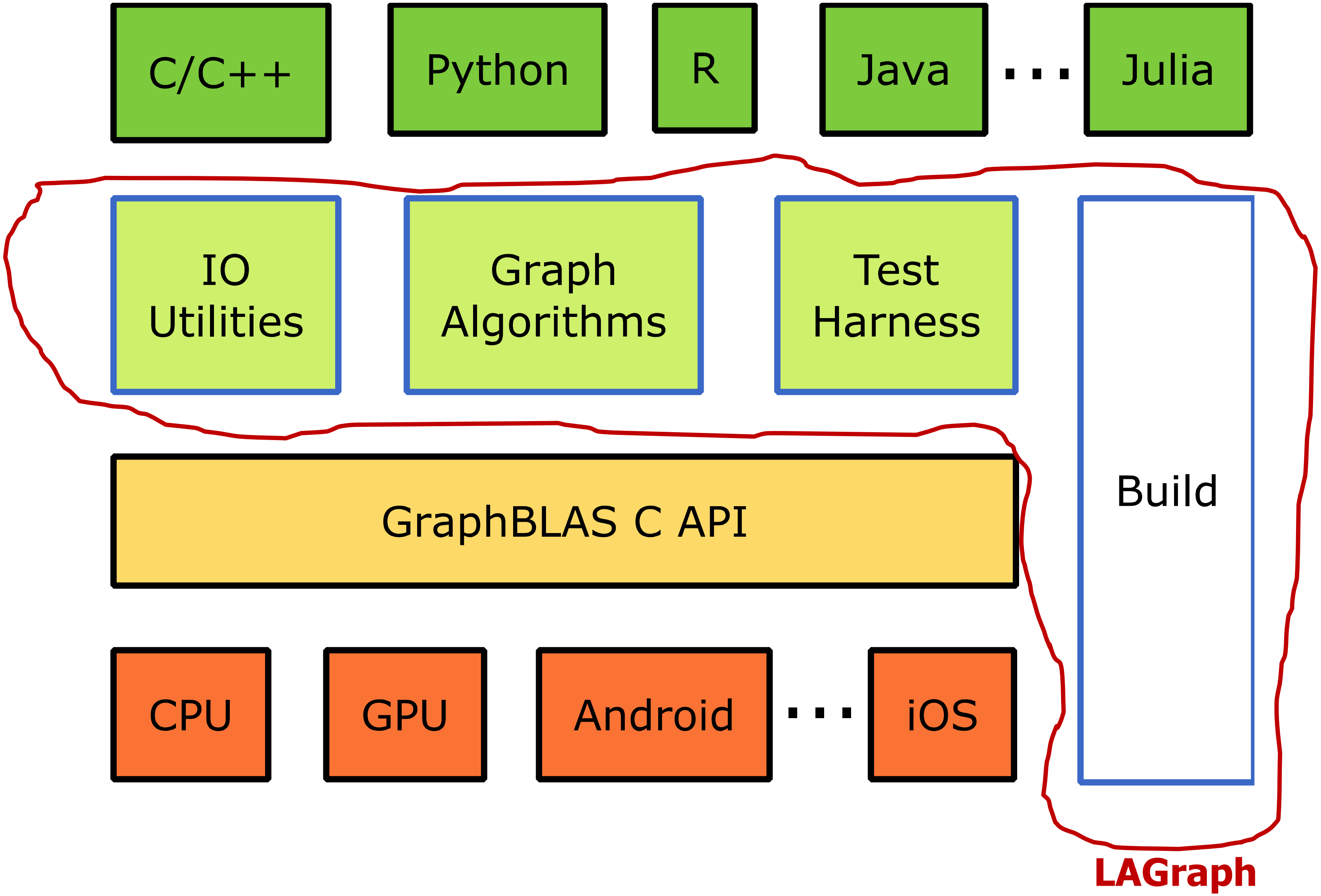
GABB@IPDPS 2017

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

GraphBLAS Implementations

- SuiteSparse library (Texas A&M): First 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/PNNL): Pushing GraphBLAS into C++
  - https://github.com/cmu-sei/gbtl
- GraphBLAST: A C++ implementation for GraphBLAS for GPUs (UC Davis),
  - https://github.com/gunrock/graphblast
- Python bindings:
  - pygraphblas: A Python Wrapper around SuiteSparse GraphBLAS
    - https://github.com/Graphegon/pygraphblas
  - grblas: Python wrapper around GraphBLAS (part of Anaconda's MetaGraph work)
    - https://github.com/metagraph-dev/grblas
  - 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/pggraphblas
- Matlab and Julia wrappers around SuiteSparse GraphBLAS
  - https://aldenmath.com

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

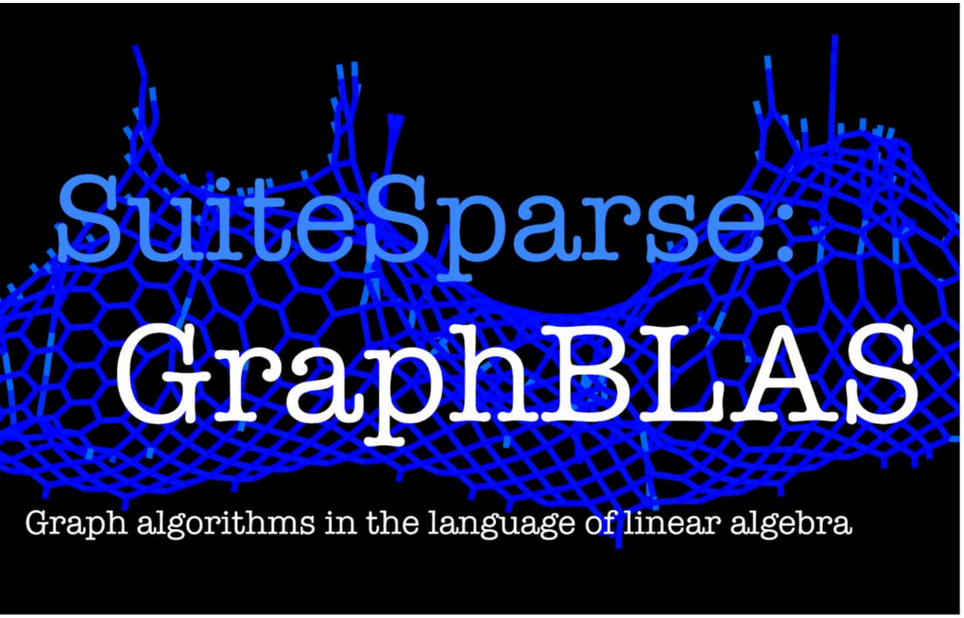
Tim Mattson[‡], Timothy A. Davis[◇], Manoj Kumar[¶], Aydın Buluç[†], Scott McMillan[§], José Moreira[¶], Carl Yang[*,†]

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

GrAPL 2019

Official release of LAGraph library v1.0 at GrAPL’21 in May 2021

SuiteSparse: C libraries for GraphBLAS and LAGraph



Tim Davis
Texas A&M
University

SuiteSparse:GraphBLAS : open-source GraphBLAS library with OpenMP (Apache 2.0)

- high performance, internal parallelism, allows for easy-to-code fast graph algorithms
- fully compliant with v1.3 C API
- MATLAB interface, many overloaded operators and functions (C(M)=A*B, etc)
- GxB extensions: ANY monoid, import/export, positional ops in semirings, scalars, float and double complex, select operation, PAIR operator, type query, subassign, ...
- matrix data structures: sparse (CSR/CSC) / hypersparse / bitmap / full
- https://people.engr.tamu.edu/davis/GraphBLAS.html

logo: mathematical art by T. D. http://www.notesartstudio.com/sincere.html

- pygraphblas is developed by Graphegon.
- Open-source package using the SuiteSparse:GraphBLAS library.
- Specializing in GraphBLAS solutions using C, Python and PostgreSQL.
- https://github.com/Graphegon/pygraphblas

Your personalized docker *container powered by*

IBM **Cloud**

"You log into the cloud by ssh and magic happens that causes little people floating in the cloud to create an account for you that runs on fairy dust and makes a container float up from the ether to respond to your every whim."

GraphBLAS in the cloud: Setting up your session

- SSH into the host `graphblas.tk` with the username `user` and password `graphblastutorial2021`.

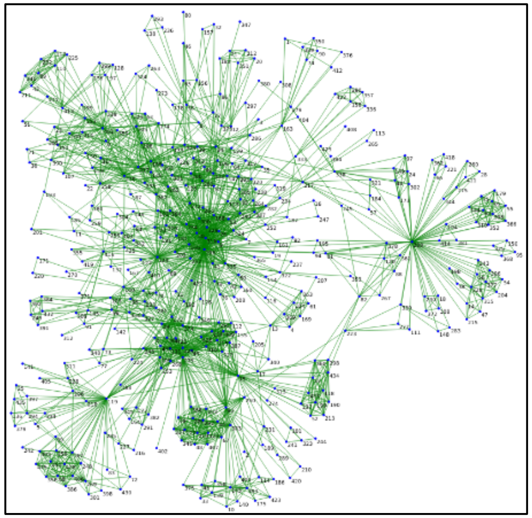
```
$ ssh user@graphblas.tk
user@graphblas.tk's password:
```

- You should see output similar to the following:

```
[I 21:42:32.806 NotebookApp] Writing notebook server cookie secret to
/home/jovyan/.local/share/jupyter/runtime/notebook_cookie_secret
[I 21:42:33.440 NotebookApp] JupyterLab extension loaded from /opt/conda/lib/python3.8/site-packages/jupyterlab
[I 21:42:33.440 NotebookApp] JupyterLab application directory is /opt/conda/share/jupyter/lab
[I 21:42:33.443 NotebookApp] Serving notebooks from local directory: /home/jovyan/SIAM-Tutorial
[I 21:42:33.443 NotebookApp] Jupyter Notebook 6.1.6 is running at:
...
Your Docker container should now be ready at:
=====
    http://graphblas.tk:98765/?token=1234567890abcdef1234567890abcdef1234567890abcdef
=====
****SAVE THIS URL!**** You will also need it for Day 2 of the tutorial.

If you do not see a URL, or there otherwise appears to be an error, please alert the tutorial staff.
Connection to graphblas.tk closed.
```

- To access the notebook, open this file in a browser (Chrome or Firefox recommended):
- file:///home/jovyan/.local/share/jupyter/runtime/nbserver-7-open.html
- Or copy and paste the **graphblas.tk URL** (above) into your browser. This will open up a Jupyter Notebook running inside a Docker container created just for you in the cloud.
- **Bookmark or save your URL.** You will need it for the second part of the tutorial tomorrow

Exercise 1: Running a GraphBLAS program



- Launch the GraphBLAS container in the cloud

  $ ssh user@graphblas.tk

- It will ask for a password

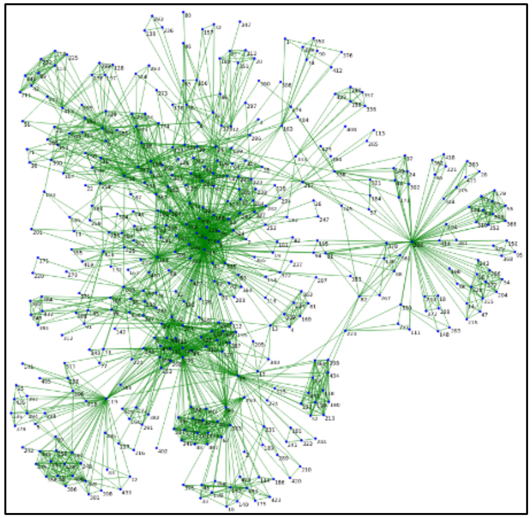
  graphblastutorial2021

- Cut-and-paste the returned URL in your browser (Save this URL so you can reconnect to the container later).

- Launch the AnalyzeGraph notebook and run-all from the cells menu to find the “2-hop” neighborhood of one author in the HPEC papers graph and then perform PageRank and draw a visualization of this reduced neighborhood [*this visualization* ***could take a while*** *to render*].

- If all goes well, we will have confirmed that everything is working … that you have container you can access through Jupyter notebook.

HPEC Authors Dataset



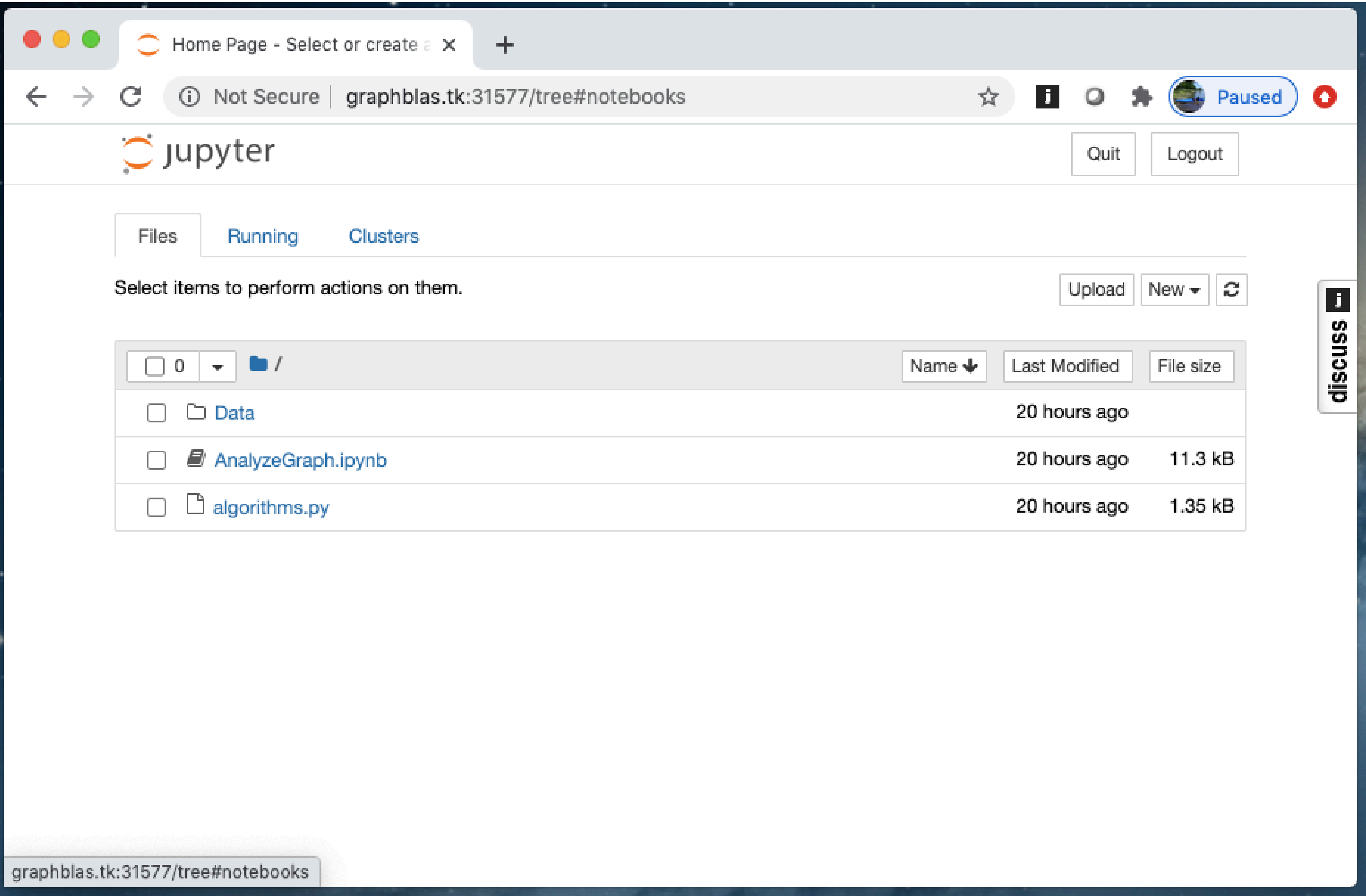
- Data represents all pairs of HPEC* authors that have coauthored papers. The edge value represents how many papers the pair have coauthored.

- Graph (undirected):
  - 1,747 vertices (unique authors)
  - 10,072 edges (coauthor count)

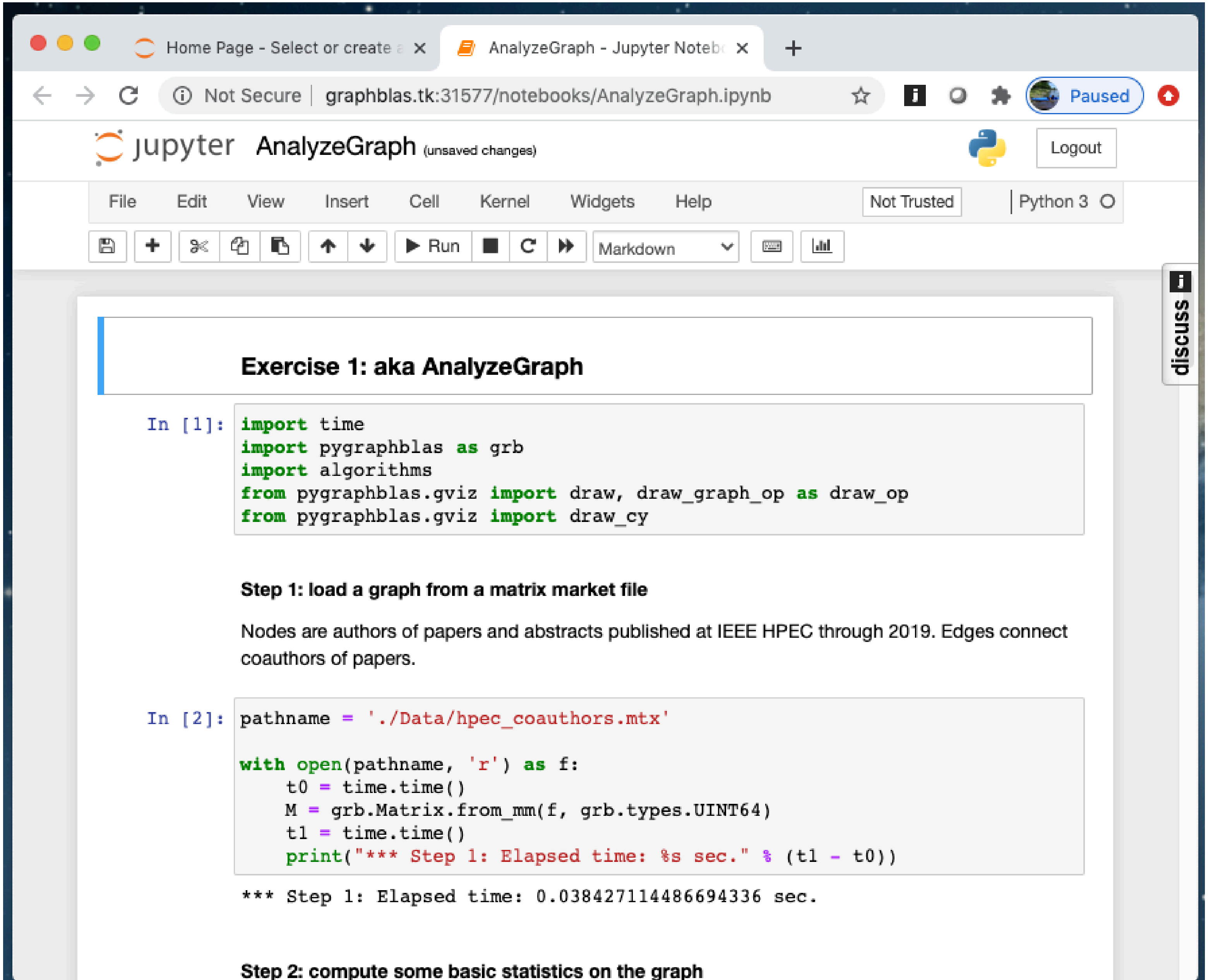
- Data directory contains index tables containing the mapping between vertex ID and author name, the raw publication data, and python scripts to perform various queries

*HPEC: IEEE High Performance Extreme Computing Conference. A conference held each September in Waltham MA where many Graph People gather each year.

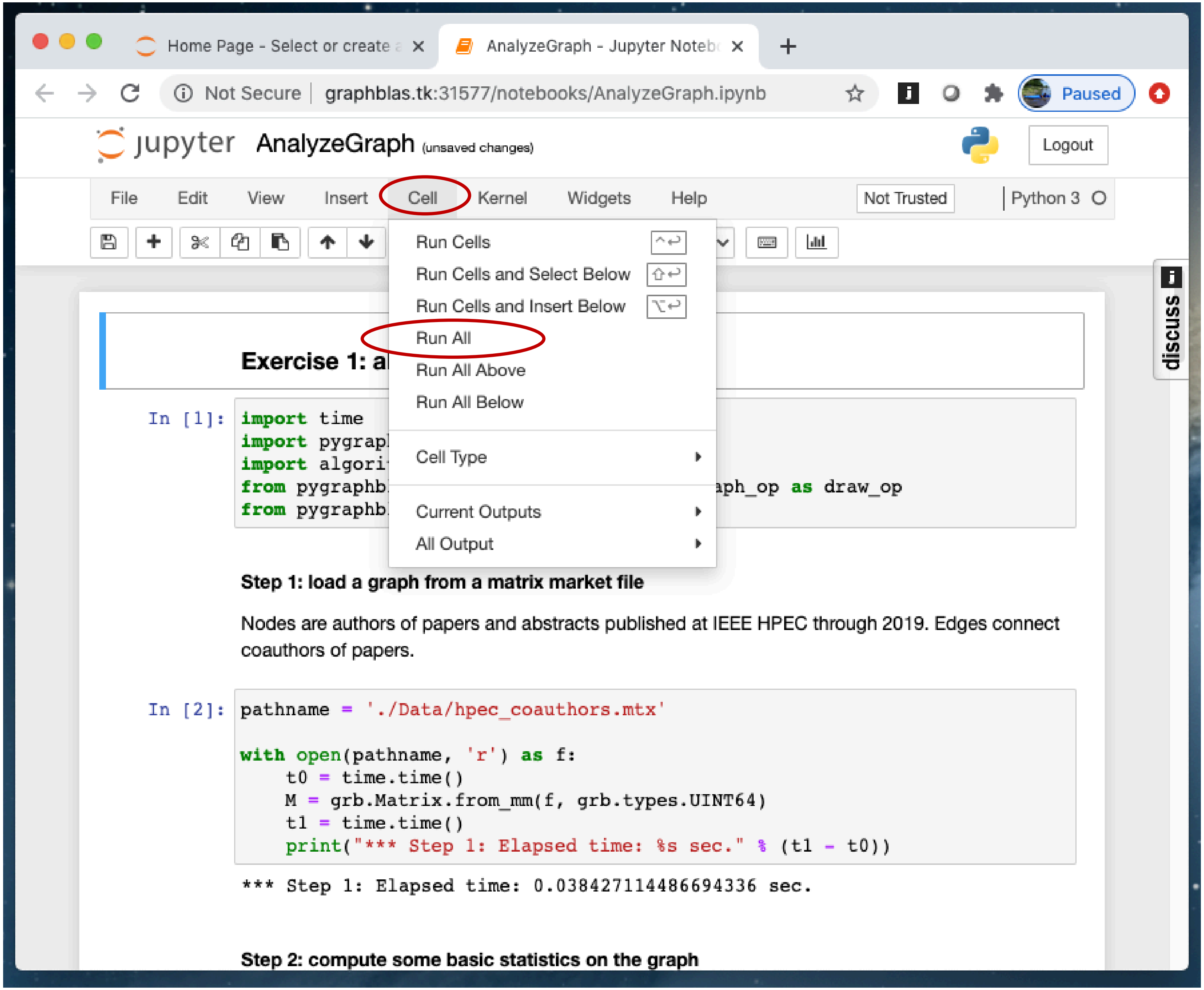
The jupyter session exposed by the container



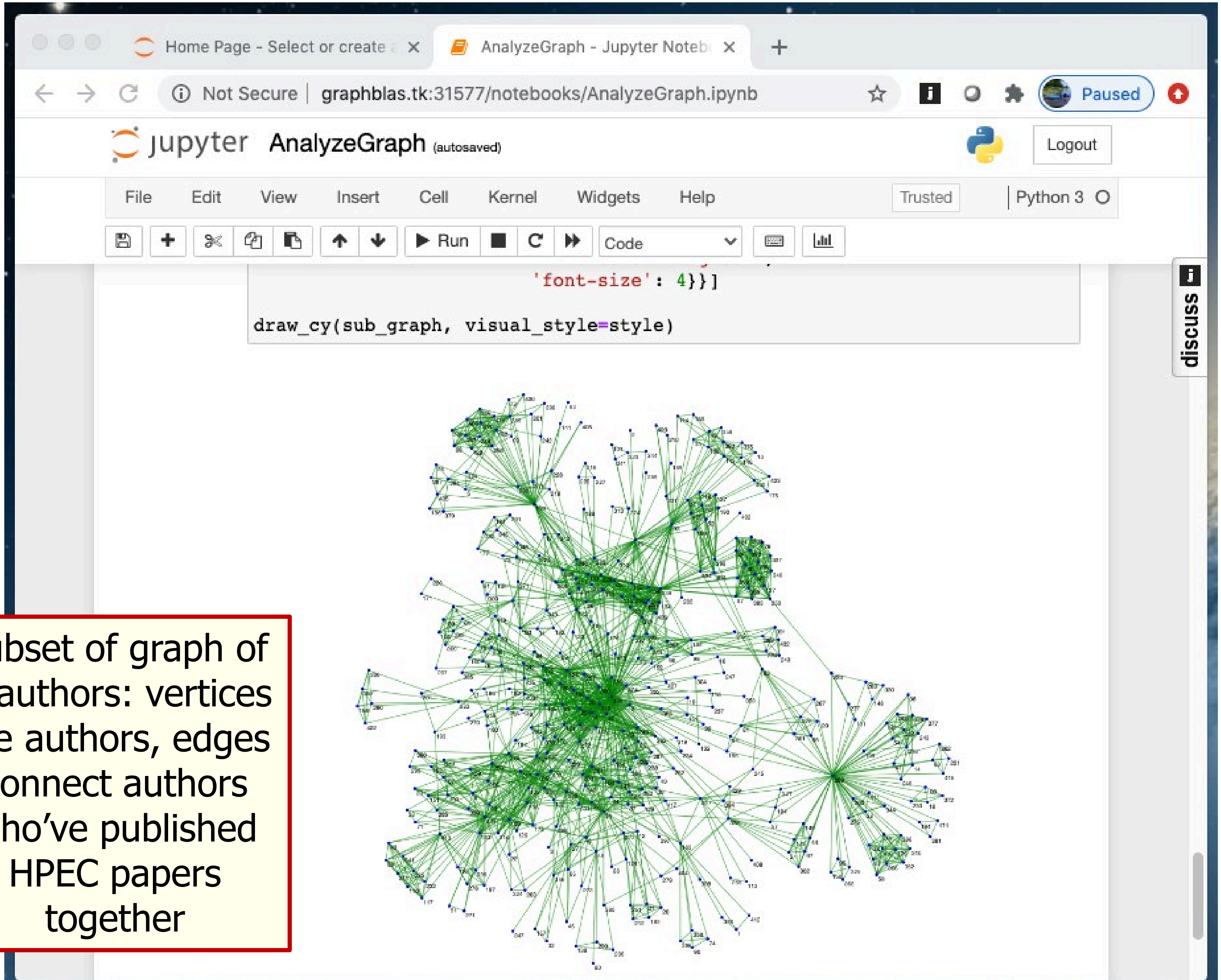
The AnalyzeGraph Notebook



Run all the cells in the notebook



The result (after a few minutes)



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

Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - Breadth-First Traversal
  - Connected Components

The GraphBLAS API:

Maps Math Spec onto the C programming language

- **Opaque object**: An object manipulated strictly through the GraphBLAS API whose implementation is not defined by the GraphBLAS specification.

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

`GrB_Vector` → A 1D sparse array

- **Method**: Any 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

```
GrB_mxv(C, GrB_NULL, GrB_NULL, GrB_LOR_LAND_BOOL, A, B, GrB_NULL);

GrB_mxv(w, GrB_NULL, GrB_NULL, GrB_LOR_LAND_BOOL, A, v, GrB_NULL);

GrB_eWiseAdd(C, GrB_NULL, GrB_NULL, GrB_LOR, A, B, GrB_NULL);
```

The GraphBLAS API: pygraphblas
Maps the C API onto Python

- **Opaque object**: An object manipulated strictly through the GraphBLAS API whose implementation is not defined by the GraphBLAS specification.

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

`Vector.sparse` → A 1D sparse array

- **Method**: Any 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 **`UINT64, INT32, BOOL, FP32`**

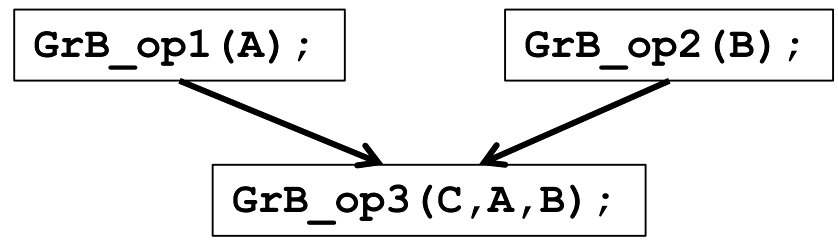
- **Operation**: a method that corresponds to an operation defined in the GraphBLAS math spec. http://www.mit.edu/~kepner/GraphBLAS/GraphBLAS-Math-release.pdf

```
Matrix multiply     C = A @ B    C = A.mxm(B)    or    A.mxm(B, out=C)
Matrix Vector       w = A @ v    w = A.mxv(v)    or    A.mxv(v, out=w)
Element-wise add    C = A + B    C = A.eadd(B)   or    A.eadd(B, out=C)
```

GraphBLAS 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 ***may*** not exit in fully defined state until queried.
- **Pygraphblas uses non-blocking mode**

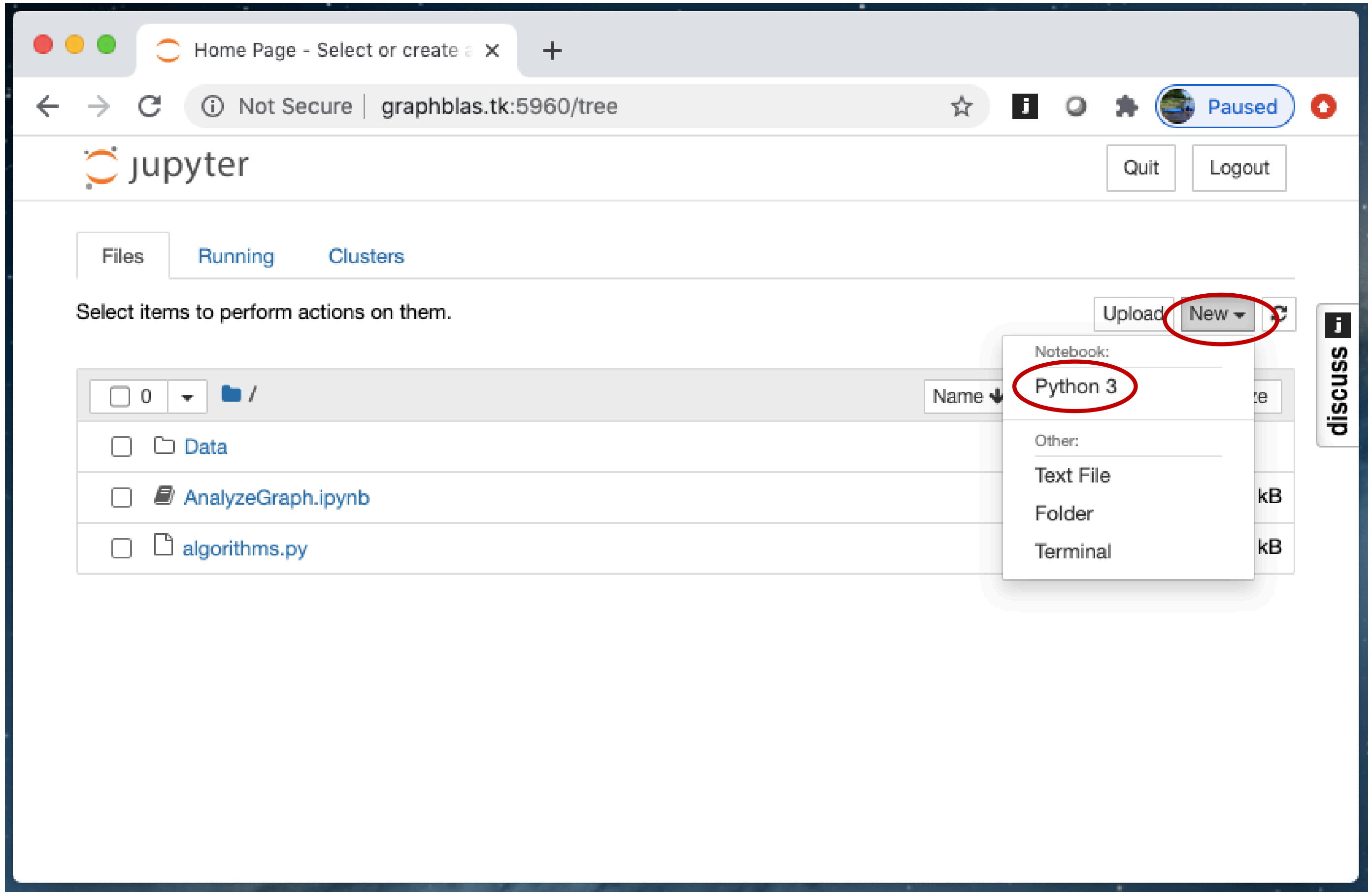
Creating a matrix

- Matrices in real problems are imported into the program from a file or an external application.

- We can build a matrix element by element
  - Import pygraphblas `import pygraphblas as grb`
  - Set size of matrix `n = 3`
  - Build a square matrix: `A = grb.Matrix.sparse(grb.UINT64,n,n)`
  - Set a value in the matrix `A[1,2] = 4`
  - Look at the matrix `print(A)`

Exercise 2: Your first pygraphblas program

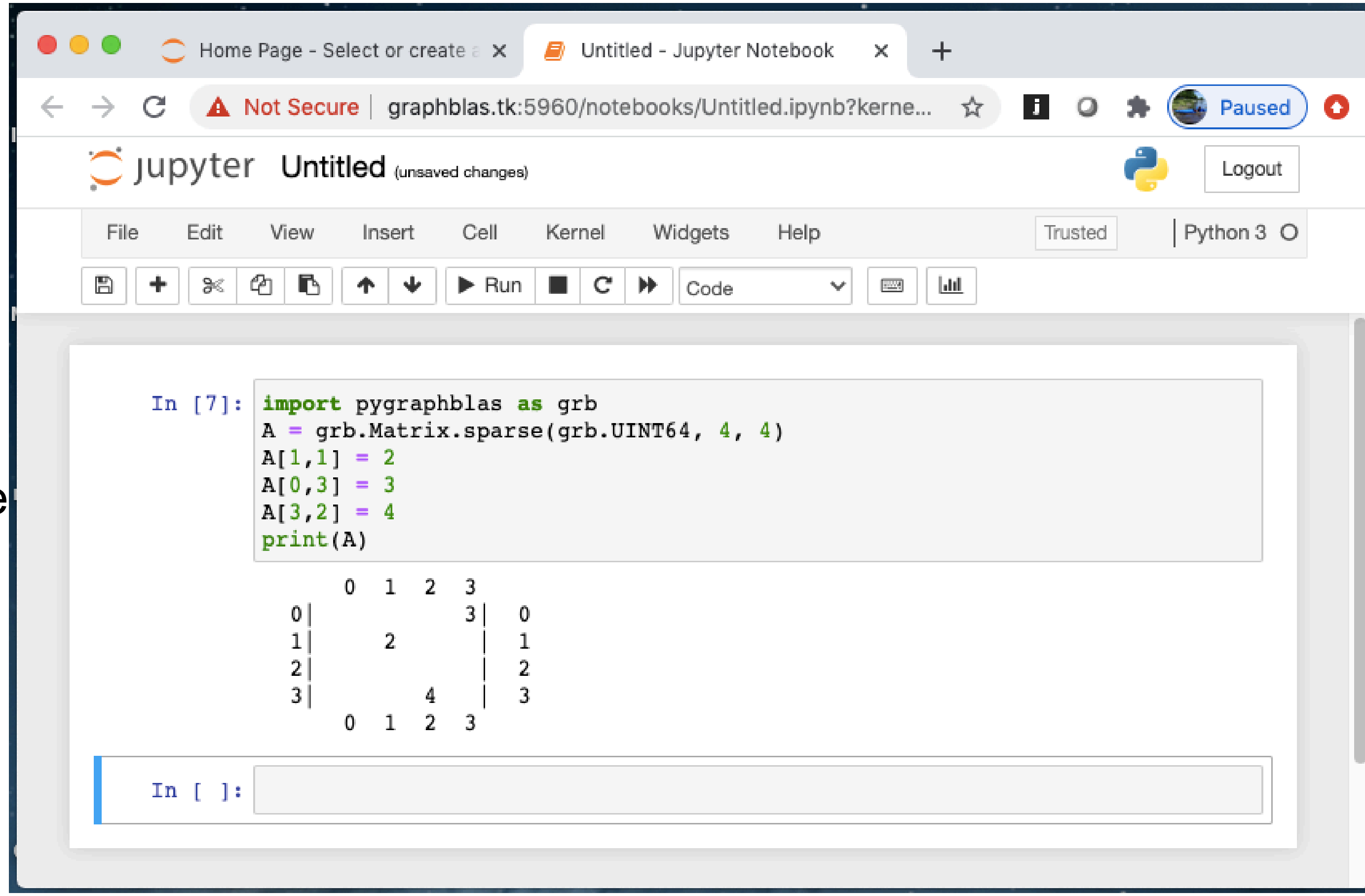
- Open a new jupyter notebook with Python 3.
- Create a matrix, set a few values, and print the result.
- Play around to make sure you are comfortable with the environment




```
import pygraphblas as grb
A = grb.Matrix.sparse(grb.type, n, n)
A[row,col] = val
some common types are UINT64, BOOL, FP32, INT8, FP64
print(A)
```

Exercise 2: Your first pygraphblas program

- Open a new jupyter notebook with Python 3.
- Create a matrix, set a few values, and print the result.
- Play around to make sure you are comfortable with the environment




```
import pygraphblas as grb
A = grb.Matrix.sparse(grb.type, n, n)
A[row,col] = val
```

some common types are UINT64, BOOL, FP32, INT8, FP64

```
print(A)
```

Creating a matrix

- Matrices in real problems are imported into the program from a file or an external application.

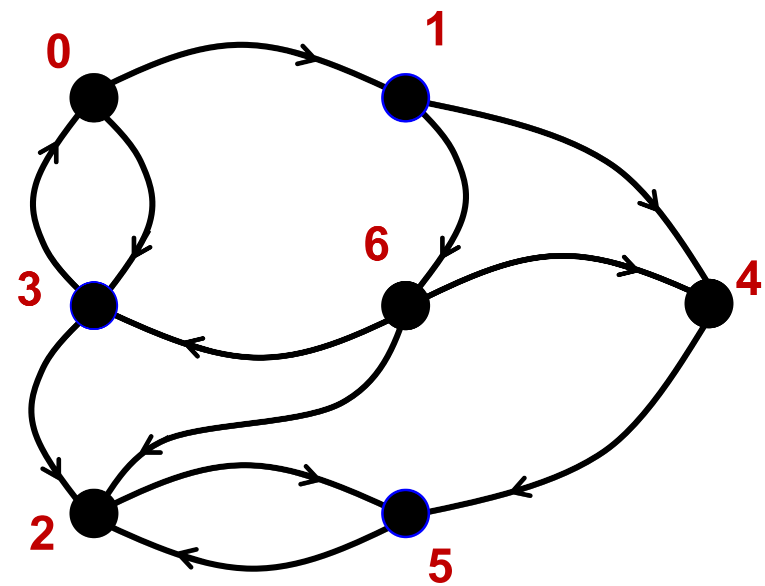
- We can build Matrices from lists:
  - Import pygraphblas: `import pygraphblas as grb`
  - List of row indices: `ri  = [0, 2, 4, 5]`
  - List of column indices: `ci  = [1, 3, 5, 5]`
  - List of Values: `val = [True]*4` ← this just creates a list of length 4
  - Build the matrix: `A = grb.Matrix.from_lists(ri, ci, val)`
  - Number of columns: `NUM_NODES = A.ncol`

- We can look at a matrix as a matrix (with print) or as a graph:

```
from pygraphblas.gviz import draw_graph as draw
draw(A)
```

Exercise 3: Adjacency matrix

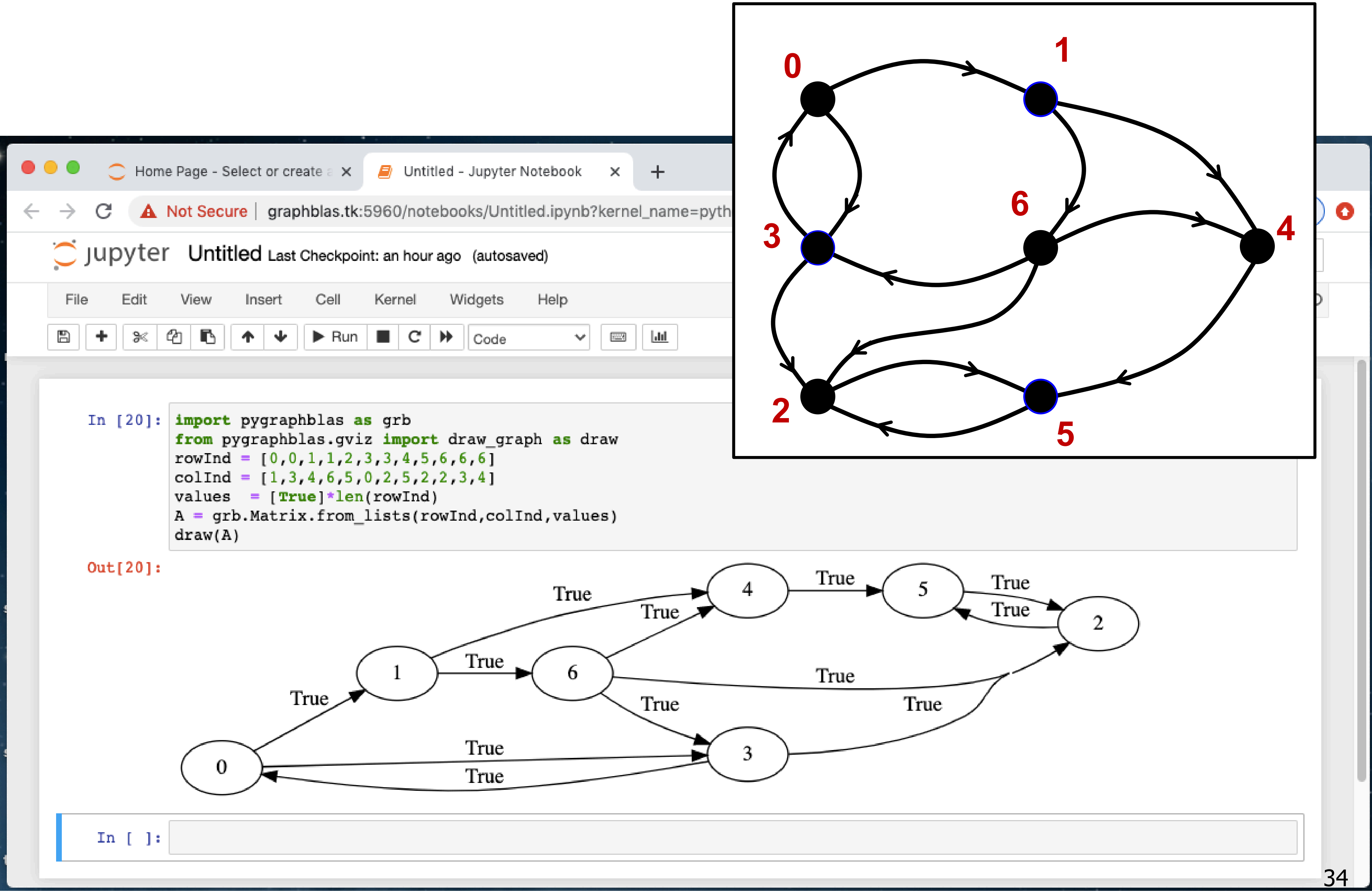
- Look at the matrix from Exercise 2 as a graph using draw_graph().
- Experiment with setting different elements until you are comfortable with the connection between a graph and an adjacency matrix.
- Create the adjacency matrix of the "GraphBLAS logo graph" from lists



- Items you will need from the pygraphblas

```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, valueList)
A = grb.Matrix.sparse(type, n, n)
A[row,col] = val
print(A)
draw(A)
some common types are UINT64, BOOL, FP32, INT8, FP64
```

Exercise 3: Adjacency matrix



Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - Breadth-First Traversal
  - Connected Components

GraphBLAS vectors and matrices

- Let's review our Matrix and Vector Objects.
- They are opaque … the structure is not defined in the spec so implementors have maximum flexibility to optimize their implementation
- pygraphblas defines a number of static methods to use when working with matrices and vectors

We've seen these already

```
import pygraphblas as grb

#Create a sparse matrix with Nrows and Ncols
A = grb.Matrix.sparse(type, Nrows, Ncols)

#Create a sparse matrix from index and value lists
A = grb.Matrix.from_lists(rowInd, colInd, valList)
```

The vector case is analogous to the matrix case

```
#Create a sparse vector of size N
v = grb.Vector.sparse(type, N)

#Create a sparse vector from index and value lists
v = grb.Vector.from_lists(indList, valList)
```

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 we can change the operators mapped onto those symbols.

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

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

mxv()

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

Multiply a matrix times a vector to produce a vector

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

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

Matrix vector multiplication: mxv()

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

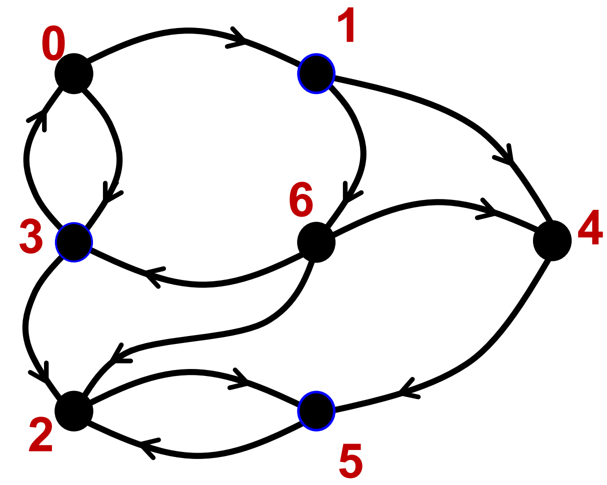
- The operators used are an accumulator ($\odot$) and the algebraic semiring operators ($\oplus$ and $\otimes$). We will say a great deal more about semirings later … for now we'll use the default case for Boolean data, the LOR_LAND semiring (i.e., logical OR for $\oplus$, Logical AND for $\otimes$).

```
import pygraphblas as grb

# A's type taken from values
A = grb.Matrix.from_lists(rowInd, colInd, values)
N = A.ncols

# Create a vector of length N and type BOOL
u = grb.Vector.sparse(grb.BOOL, N)
u[ind] = True     # set the value

w = A.mxv(u)

w = A @ u

A.mxv(u, out=w)
```

These three compute the same result in w.
The third reuses an existing w.

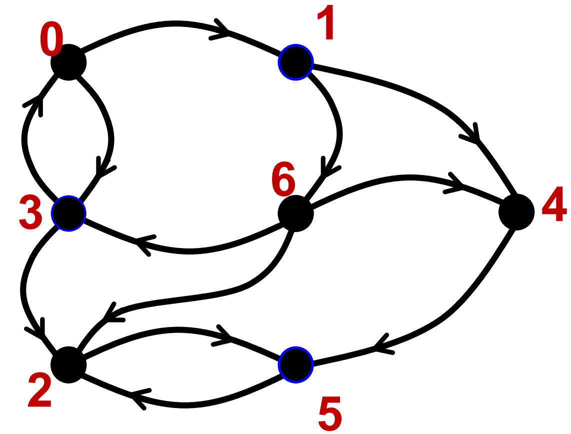
Exercise 4: Matrix Vector Multiplication

- 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 **<u>interpret its meaning</u>**.
- You'll need the following from pygraphblas:




```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.BOOL, N)
w = A.mxv(u)
A.mxv(u, out=w)    # reuse a w that already exists
w = A @ u
print(A), print(w)
draw(A)
```

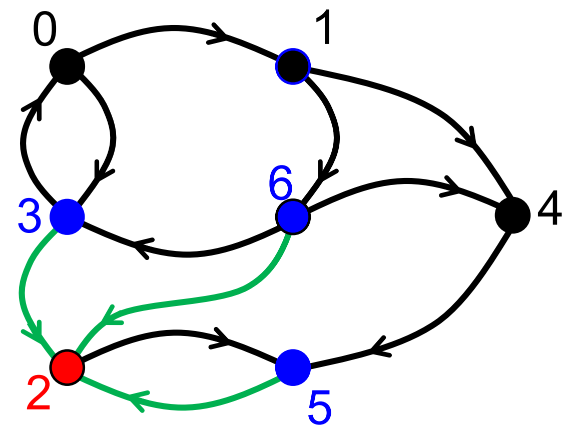
Solution to exercise 4




```
NODES = 7
u = grb.Vector.sparse(grb.BOOL, NODES)
u[2] = True
w = A @ u
print(w)
```

```
w                        A                          u

                   0   1   2   3   4   5   6
0|           0|        t       t            |  0          0|
1|           1|                    t       t|  1          1|
2|           2|                        t    |  2          2|  t
3|  t   =    3|    t       t                |  3    @     3|
4|           4|                        t    |  4          4|
5|  t        5|            t                |  5          5|
6|  t        6|            t   t   t        |  6          6|
                   0   1   2   3   4   5   6
```

Solution to exercise 4




```
Nodes = 7
u=grb.Vector.sparse(grb.BOOL,Nodes)
u[2] = True
w=A@u
print(w)
```

```
   w                        A                           u

                  0  1  2  3  4  5  6
   0|         0|     t     t          |  0          0|
   1|         1|              t     t|  1          1|
   2|         2|                 t    |  2          2|  t
   3|  t   =  3|  t     t             |  3    @     3|
   4|         4|                 t    |  4          4|
   5|  t      5|        t             |  5          5|
   6|  t      6|        t  t  t       |  6          6|
                  0  1  2  3  4  5  6
```

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)

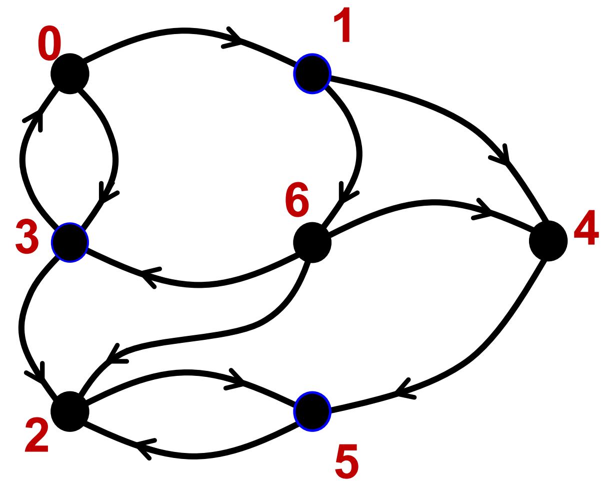
$$\text{neighbors} = A^T \oplus . \otimes v$$

Two ways using pygraphblas

```
neighbors = A.T @ v
neighbors = A.mxv(v, desc=grb.descriptor.T0)
```

Exercise 5: Find one hop neighbors

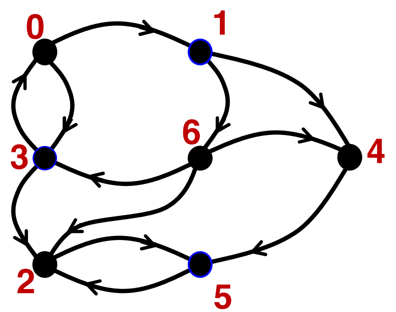
- This is a really quick exercise.
- Go back to your code for exercise 4 and verify that you can use the transpose to find the one hop neighbors of a source vertex.




```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.BOOL, N)
Atrans = A.T
w = A.mxv(u)
A.mxv(u, out=w)
w = A @ u
print(A)
draw(A)
```

Solution to exercise 5: Find one-hop neighbors

```
NODES = 7
u = grb.Vector.sparse(grb.BOOL, NODES)
u[6] = True
w = A.T @ u
print(w)
```



A

```
    0  1  2  3  4  5  6
 0|    t     t         |  0
 1|             t     t|  1
 2|                t   |  2
 3| t     t            |  3
 4|                t   |  4
 5|       t            |  5
 6|       t  t  t      |  6
    0  1  2  3  4  5  6
```

```
w                      A^T                                u

                       0  1  2  3  4  5  6
0|                  0|          t          |  0           0|
1|                  1| t                   |  1           1|
2| t                2|          t     t   t|  2           2|
3| t        =       3| t                  t|  3     @     3|
4| t                4|    t               t|  4           4|
5|                  5|       t     t       |  5           5|
6|                  6|    t                |  6           6| t
                       0  1  2  3  4  5  6
```

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 only covered mxv, so far, but the same conventions are used across all operations, so we'll have no problem later when we use the others

<M, m>: write masks (Matrix or vector).
<z>: selects "replace or combine" for elements not selected by the mask.

Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - Breadth-First Traversal
  - Connected Components

pygraphblas:

- A python wrapper around the SuiteSparse GraphBLAS library.
  - Brought to us by Michel Pelletier and his company Graphegon.

- pygraphblas uses CFFI to automate much of the process of generating an interface to SuiteSparse GraphBAS library … thus helping pygraphblas to closely track new releases of SuiteSparse.

- Open-Source release … also provided as docker containers:
  - Minimal: an Ipython interpreter-only
  - Notebook: Includes a Jupyter notebook server

- Documentation for pygraphblas (with lots of examples) can be found here:
  - https://graphegon.github.io/pygraphblas/pygraphblas/index.html

pygraphblas: Matrix and Vector part 1

- Matrix constructors … commonly used cases:
  - **sparse(typ, nrows=None, ncols=None)**
  - **dense(typ, nrows, ncols, fill=None, sparsity=None)**
  - **from_lists(I, J, V, nrows=None, ncols=None, typ=None)**
  - **dup(A)**
  - **random(typ, nrows, ncols, nvals, make_pattern=False,**
    make_symmetric=False, make_skew_symmetric=False,
    make_hermitian=True, no_diagonal=False, seed=None)

Common types (typ) are: BOOL, FP64, FP32, INT64, INT32, INT16, INT8, UINT64, UINT32, UINT16, UINT8

- Vector constructors … commonly used cases:
  - **sparse(typ, size=None)**
  - **from_lists(I, V, size=None, typ=None)**
  - **from_1_to_n(n)**
  - **dense(typ, size, fill=None)**
  - **dup(v)**

For arguments with default value **None**, if the argument is not provided, pygraphblas will infer what it needs from the other arguments

```
import pygraphblas as grb
N = 8
Nval = 16
graph = grb.Matrix.random(grb.INT8,N,N,Nval,make_symmetric=True)
g2 = graph.dup()
vec = grb.Vector.from_1_to_n(N)
res = g2@vec
```

pygraphblas: Matrices and Vectors part 2

- Matrix with Instance attributes and properties … a few of which are:
  - **nrows**
  - **ncols**
  - **nvals**
  - **T ← take the transpose of the matrix**
  - **M ← create a bool matrix, true where a defined element exists**
  - **get(i,j,default=None)**
- Vector with Instance attributes and properties … a few of which are:
  - **size**
  - **nvals**
  - **Indexes**
  - **get(i, ,default=None)**

```
import pygraphblas as grb
g2 = grb.Matrix.from_lists([1,1,2],[1,2,0],[4,-5,0])
gm = graph.M
g4 = grb.Matrix.mxm(g2, g2.T, mask=gm)
NUM_NODES = g2.nrows
vec = grb.Vector.from_1_to_n(NUM_NODES)
for i in vec.indexes:
    do_something(i)
jj = vec.size/2
print(vec.get(jj,default=10))
```

Uses a write-mask to select elements of the product to store

Using an iterator over the vector

Extract element jj. If there is no element jj, then return the default value 10 (the default default is None)

Changing the behavior of a GraphBLAS operation

- Most GraphBLAS operations take an argument that is an opaque object called a “descriptor”.
- The descriptor controls the behavior of the method and how objects are handled inside the method.
- The descriptor controls:
  - Do you *transpose input matrices*?
    - T0 ← transpose first argument
    - T1 ← transpose second argument

They can be combined:
T0T1 ← transpose both args

  - Does the computation *replace existing values in the output object* or combine with them when a mask is used?
  - Take the *structure* and/or *complement* of the *mask* object (swap empty/false ←→ filled/true values in a sparse object).

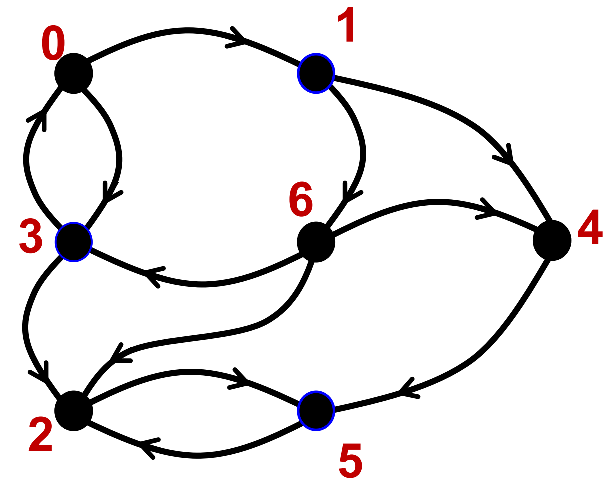
....To be discussed later

Descriptor example: `Hop = A.mxm(B, desc = grb.descriptor.T0)`

A pygraphblas descriptor is a distinct module so you must specify it as such when used.
In this example, T0 transposes A. T1 would transpose B

Exercise 6: find one hop neighbors (again)

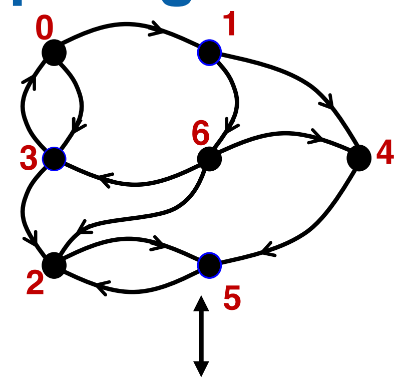
- This is a really quick exercise.
- Go back to your code for exercise 5 and verify that you can specify a descriptor to use the transpose to find the one hop neighbors of a source vertex.




```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.BOOL, N)
Atrans = A.T
w = A.mxv(u, desc = ??)
A.mxv(u, out=w, desc = ??)
print(A)
draw(A)
```

Solution to exercise 6: Find one-hop neighbors

```
NODES = 7
u = grb.Vector.sparse(grb.BOOL, NODES)
u[6] = True
w = A.mxv(u, desc = grb.descriptor.T0)
print(w)
```




```
      0  1  2  3  4  5  6
   0|    t     t          | 0
   1|             t     t | 1
   2|                t    | 2
A  3| t     t             | 3
   4|                t    | 4
   5|       t             | 5
   6|       t  t  t       | 6
      0  1  2  3  4  5  6
```

```
                              A^T
 w                                                      u
                  0  1  2  3  4  5  6
 0|           0|           t        |  0           0|
 1|           1|  t                 |  1           1|
 2|  t        2|           t     t t|  2           2|
 3|  t    =   3|  t                t|  3     @     3|
 4|  t        4|     t             t|  4           4|
 5|           5|        t     t     |  5           5|
 6|           6|     t              |  6           6|  t
                  0  1  2  3  4  5  6
```

Matrix vector multiplication: mxv()

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

It's time to say more about these operators

- The operators in GraphBLAS operations are:
  - Accumulator $\odot$
  - Addition $\oplus$
  - Multiplication $\otimes$
- The operators are implied by type.

Pygraphblas type	$\odot$	$\oplus$	$\otimes$
types.BOOL	LOR	LOR	LAND
types.INT32	32 bit int add	32 bit int add	32 bit int multiply
types.FP32	32 bit float add	32 bit float add	32 bit float multiply

```
import pygraphblas as grb

# Matrix A and vectors w and u defined elsewhere.
# Assume they are of type INT32
A.mxv(u,accum=grb.INT32.PLUS,out=w)
```

Implicit $\oplus$ and $\otimes$ based on type of A, u, and/or w. $\odot$ defined explicitly

$\oplus . \otimes$ are considered together as part of an algebraic semiring (our next topic)

Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing ($\oplus$,$\otimes$) 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 identity is an Op2 annihilator.

Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing ($\oplus$,$\otimes$) 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)

(R,	+,	*,	0,	1)
Scalar type	Op1	Op2	Identity Op1	Identity Op2

Algebraic Semirings

- Semiring: An Algebraic structure that generalizes real arithmetic by replacing ($\oplus$,$\otimes$) 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.

Working with semirings

- In Graph Algorithms, changing semirings multiple times inside a single algorithm is quite common. Hence, the semiring (and implied accumulator $\oplus$ by default) can be directly manipulated.
- We can do this using python's with statement

```
with grb.BOOL.LOR_LAND:
    w += A@v

with grb.BOOL.LOR_LAND:
    w = A.mxv(u,accum=grb.BOOL.LOR)
```

Matrix vector product of A and v accumulating the result with the existing elements of w using:

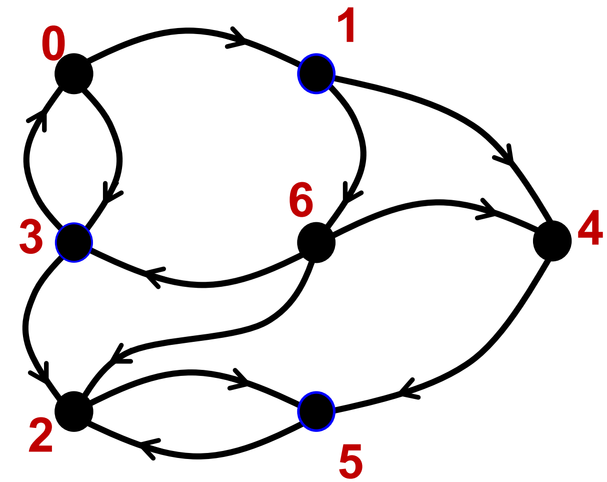
- $\oplus = \odot$ = Logical OR
- $\otimes$ = Logical AND

- Common semirings include (though pygraphblas has MANY more)

(R, +, *, 0, 1) Real Field	Standard operations in linear algebra	**FP64.PLUS_TIMES**
(R U {∞},min, +, ∞, 0) Tropical semiring	Shortest path algorithms	**FP64.MIN_PLUS**
({0,1}, |, &, 0, 1) Boolean Semiring	Graph traversal algorithms	**BOOL.LOR_LAND**
(R U {∞}, min, *, ∞, 1)	Selecting a subgraph or contracting nodes to form a quotient graph.	**FP64.MIN_TIMES**

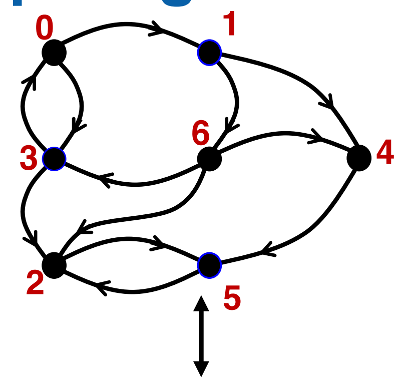
Exercise 7: find one hop neighbors (again)

- This is a really quick exercise.
- Go back to your code for exercise 6 and verify that you can specify the semiring to find the one hop neighbors of a source vertex.




```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.BOOL, N)
Atrans = A.T
with grb.<type>.<semiring>:
w = A.mxv(u, desc = ??)
A.mxv(u, out=w, desc = ??)
print(A)
draw(A)
```

Solution to exercise 7: Find one-hop neighbors




```
NODES = 7
u = grb.Vector.sparse(grb.BOOL, NODES)
u[6] = True
with grb.BOOL.LOR_LAND:
   w = A.mxv(u, desc = grb.descriptor.T0)
print(w)
```

```
      0  1  2  3  4  5  6
   0|    t     t        | 0
   1|             t     t| 1
   2|                t  | 2
A  3| t     t           | 3
   4|                t  | 4
   5|       t           | 5
   6|       t  t  t     | 6
      0  1  2  3  4  5  6
```

```
                              Aᵀ
w                                                         u
                  0   1   2   3   4   5   6
0|            0|              t           |  0          0|
1|            1|  t                       |  1          1|
2|  t         2|              t       t   t|  2          2|
3|  t    =    3|  t                       t|  3    @     3|
4|  t         4|      t                   t|  4          4|
5|            5|          t       t       |  5          5|
6|            6|      t                   |  6          6|  t
                  0   1   2   3   4   5   6
```

GraphBLAS Operations (a subset from the Math Spec*)

Name	Math	pygraphblas examples (`import pygraphblas as grb`)
mxm	$C \odot= A \oplus.\otimes B$	`mxm(self, other, cast=None, out=None, semiring=None, mask=None, accum=None, desc=None)`
mxv	$w \odot= A \oplus.\otimes v$	`mxv(self, other, cast=None, out=None, semiring=None, mask=None, accum=None, desc=None)`
vxm	$w^T \odot= v^T \oplus.\otimes A$	`vxm(self, other, cast=None, out=None, semiring=None, mask=None, accum=None, desc=None)`
eWiseMult	$C \odot= A \otimes B$ $w \odot= u \otimes v$	`emult(self, other, mult_op=None, cast=None, out=None, mask=None, accum=None, desc=None)` `emult(self, other, mult_op=None, cast=None, out=None, mask=None, accum=None, desc=None)`
eWiseAdd	$C \odot= A \oplus B$ $w \odot= u \oplus v$	`eadd(self, other, add_op=None, cast=None, out=None, mask=None, accum=None, desc=None)` `eadd(self, other, add_op=None, cast=None, out=None, mask=None, accum=None, desc=None)`
reduce	$w \odot= \oplus_j A(:,j)$	`reduce_vector(self, mon=None, out=None, mask=None, accum=None, desc=None)`

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

GraphBLAS Operations (a subset from the Math Spec*)

Name	Math	pygraphblas examples (`import pygraphblas as grb`)

mxm	$C \odot= A \oplus.\otimes B$	`A.mxm(B, out=C, accum=grb.INT64.PLUS)`
		
		`with Accum=grb.INT64.PLUS:`
		`    C = A@B`

mxv	$w \odot= A \oplus.\otimes v$	`with grb.INT64.PLUS_TIMES:` `    w = A@v`
vxm	$w^T \odot= v^T \oplus.\otimes A$	`w = v.vxm(A, mask=None, accum=grb.FP32.PLUS)`

eWiseMult	$C \odot= A \otimes B$
	
	$w \odot= u \otimes v$
		`A.emult(B, out=C, accum=grb.INT32.PLUS)`
		
		`w=u*v`

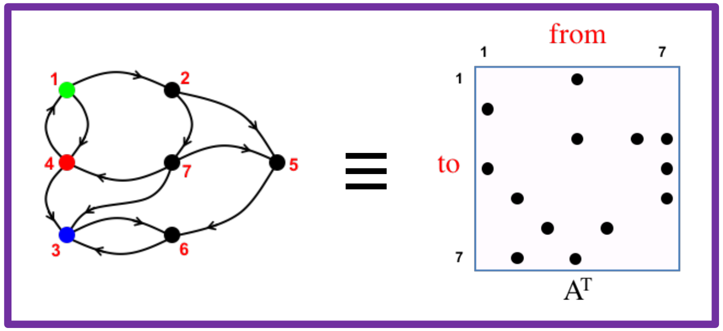
eWiseAdd	$C \odot= A \oplus B$
	
	$w \odot= u \oplus v$
		`C = A+B`
		
		`w=y.eadd(v, add_op=grb.FP64.DIV)`

reduce	$w \odot= \oplus_j A(:,j)$	`A.reduce_vector(out=w, accum=grb.FP32.PLUS)`

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

Pygraphblas Documentation

- We have only covered the common cases we work with in this tutorial.
- The pygraphblas documentation describes:
  - Numerous additional semirings organized by type.
  - Numerous binary and unary functions so you can choose your own accumulators or build custom semirings.
  - The ability to create custom binary and unary operators using a JIT decorator.
  - Options to control multithreading and other internal features of the SuiteSparse GraphBLAS library
  - The pygraphblas.gviz sub-module which includes methods for looking at small graphs but also a binding to the Cytoscape module for working with large complex graphs



A Hands-On Introduction to GraphBLAS:
The Python Edition (DAY 2)

http://graphblas.org

Scott McMillan
CMU/SEI

Tim Mattson
Intel Labs

… and the other members of the GraphBLAS specification group:
Aydın Buluç (UC Berkeley/LBNL), Jose Moreira (IBM), and **Ben Brock (UC Berkeley).**

With a special thank you to **Tim Davis (Texas A&M)** for GraphBLAS support in SuiteSparse and **Michel Pelletier (Graphegon)** for creating pygraphblas.

To get course materials onto your laptop:

```
$ git clone -b classroom21 https://github.com/GraphBLAS/SIAM-Tutorial.git
```

Reminders

To get course materials onto your laptop (updated README, slides):

```
$ git clone -b classroom21 https://github.com/GraphBLAS/SIAM-Tutorial.git
```

Reopen your tab from yesterday:

…
Your Docker container should now be ready at:
=====
http://graphblas.tk:98765/?token=1234567890abcdef1234567890abcdef1234567890abcdef
=====
******SAVE THIS URL!****** You will also need it for Day 2 of the tutorial.

If you do not see a URL, or there otherwise appears to be an error, please alert the tutorial staff.
Connection to graphblas.tk closed.

If you lost your URL, you can spin up a new server (code on next page):

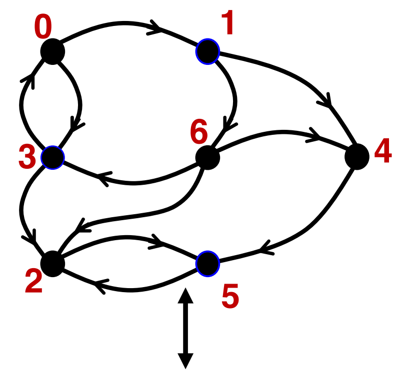
$ ssh user@graphblas.tk
user@graphblas.tk's password: **graphblastutorial2021**

Documentation for pygraphblas (with lots of examples) can be found here:

https://graphegon.github.io/pygraphblas/pygraphblas/index.html

Where we left off: Solution to exercise 7:

Finding one-hop *out*-neighbors




```
       0  1  2  3  4  5  6
   0|     t     t        |  0
   1|              t     t|  1
   2|                 t  |  2
A  3|  t     t           |  3
   4|                 t  |  4
   5|        t           |  5
   6|        t  t  t     |  6
       0  1  2  3  4  5  6
```

```
import pygraphblas as grb

rowID = [0,0,1,1,2,3,3,4,5,6,6,6]
colID = [1,3,4,6,5,0,2,5,2,2,3,4]
vals  = [True]*len(rowID)
A = grb.Matrix.from_lists(rowID, colID, vals)

u = grb.Vector.sparse(grb.BOOL, A.nrows)
u[6] = True
with grb.BOOL.LOR_LAND:
    # w = A.T @ u
    w = A.mxv(u, desc = grb.descriptor.T0)

print(w)
```

```
               A^T                             u            w
      0  1  2  3  4  5  6
   0|          t          |  0            0|           0|
   1|  t                  |  1            1|           1|
   2|          t     t   t|  2            2|           2|  t
   3|  t                 t|  3     @      3|     ➔     3|  t
   4|     t              t|  4            4|           4|  t
   5|        t     t      |  5            5|           5|
   6|     t               |  6            6|  t        6|
      0  1  2  3  4  5  6
```

Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - Breadth-First Traversal
  - Connected Components

Breadth First Traversal (aka BFS)

- 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: Need to 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 variety 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

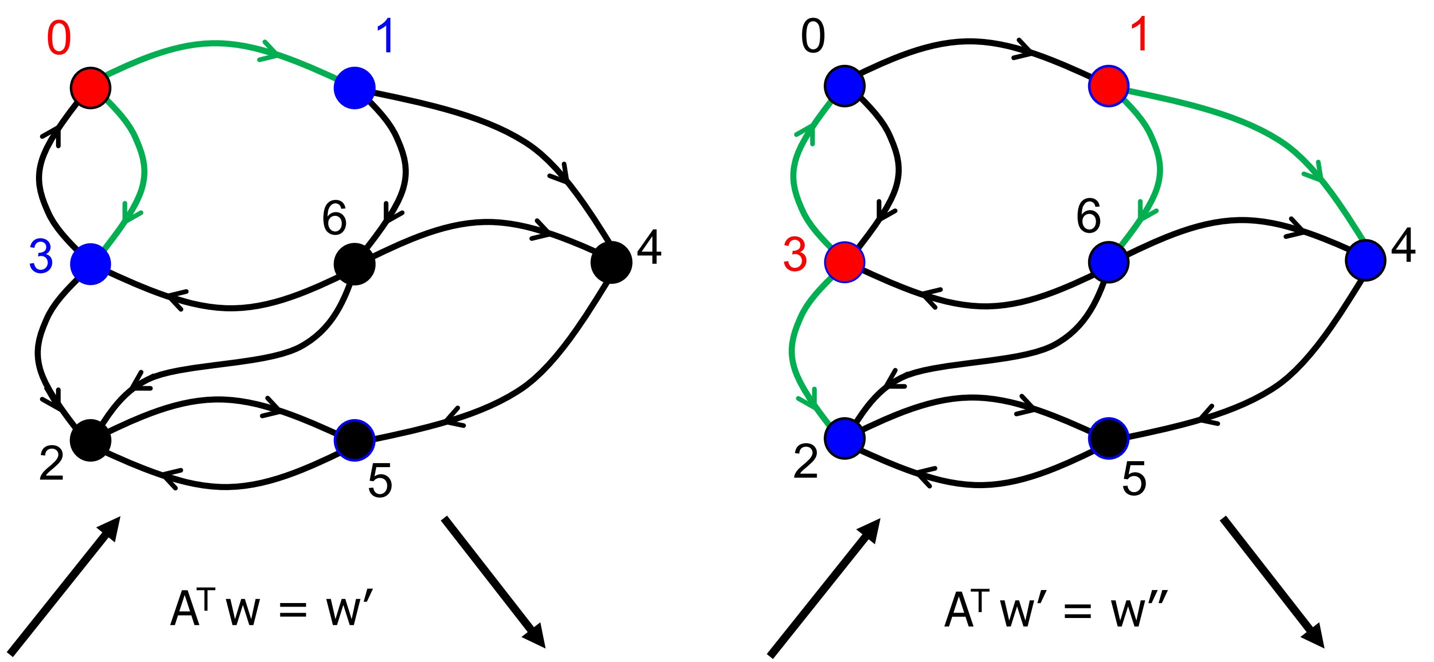
- We will build up this algorithm using the GraphBLAS through a series of exercises:
  - Wavefronts and how to move from one wavefront to the next.
  - Iteration across wavefronts
  - Track which vertices have been visited
  - Avoid revisiting vertices

*** At this point you have a basic BFS algorithm. ***

  - Use this to 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…"



$A^T w = w'$ $\quad$ $A^T w' = w''$

$w = \{0\}$ $\quad$ $w' = \{1, 3\}$ $\quad$ $w'' = \{0, 2, 4, 6\}$

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

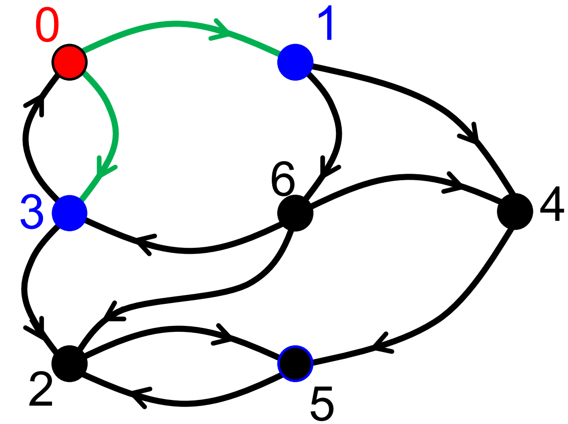
A = Adjacency Matrix $\qquad$ w = wavefront vectors

Exercise 8: Traverse the graph

- Modify your code from Exercises 5/6/7 to iterate from one wavefront to the next.
- Output each wavefront
- How long before you get a repeating pattern?

```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.<type>, N)
Atrans = A.T
with grb.<type>.<semiring>:
w = A.mxv(u, desc = ??)
A.mxv(u, out=w, desc = ??)
print(A)
draw(A)
```

Solution to exercise 8



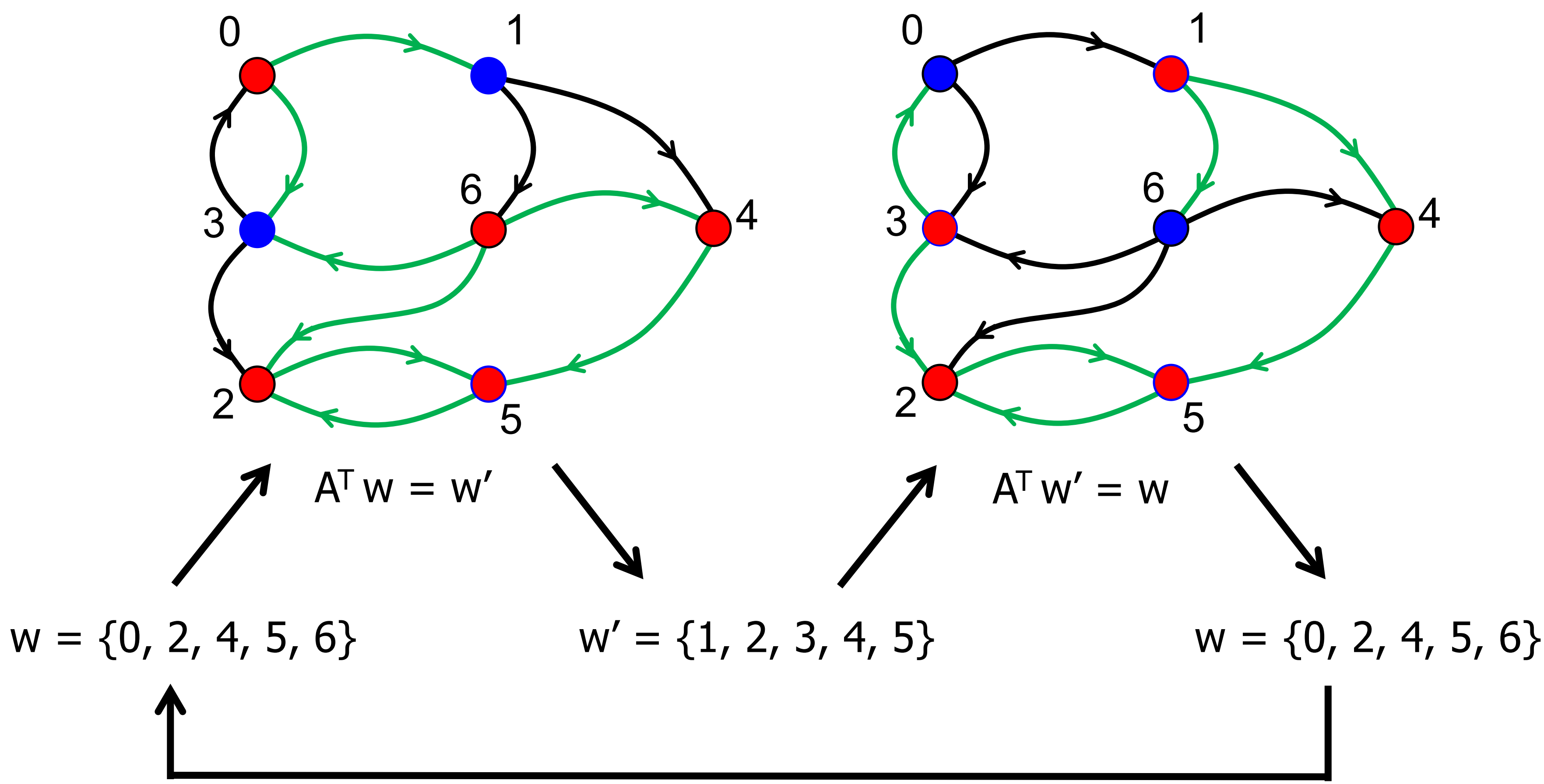

```
NUM_NODES = 7;
w = grb.Vector.sparse(grb.BOOL, NUM_NODES);
w[0] = True  # 1st wavefront has one node set.
print(w)

with grb.BOOL.LOR_LAND:
    for i in range(NUM_NODES):
        w.mxv(graph, out=w, desc=grb.descriptor.T0)
        print(w)
```

Init.	i=0	i=1	i=2	i=3	i=4	i=5	i=6
0| t	0|	0| t	0|	0| t	0|	0| t	0|
1|	1| t	1|	1| t	1|	1| t	1|	1| t
2|	2|	2| t	2| t	2| t	2| t	2| t	2| t
3|	3| t	3|	3| t	3|	3| t	3|	3| t
4|	4|	4| t	4| t	4| t	4| t	4| t	4| t
5|	5|	5|	5| t	5| t	5| t	5| t	5| t
6|	6|	6| t	6|	6| t	6|	6| t	6|

The same vector 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)"



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

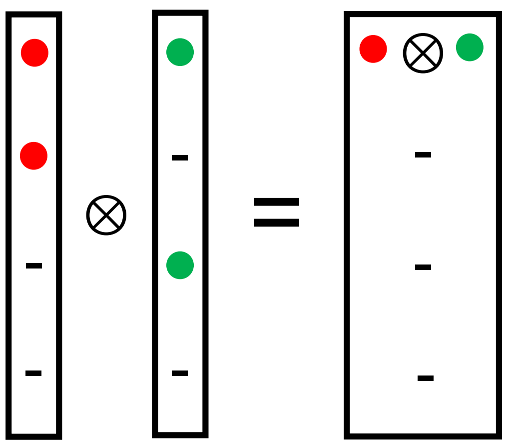
Visited lists

- Breadth-first traversal requires that we only need to visit each node once.
- First step is to keep track of visited nodes.
- You can do this by accumulating the wavefronts.
  - Use element-wise “addition” with logical-OR.

Element-wise Operations: Mult and Add

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

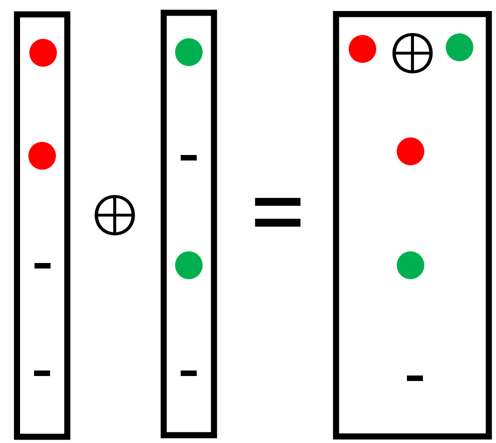
$\mathbf{u} \otimes \mathbf{v}$



Examples: (x,0), (and, false), (+, ∞)

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

$\mathbf{u} \oplus \mathbf{v}$



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

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

Element-wise Multiplication $w \cancel{\otimes}= (u \otimes v)$

- Compute the element-wise "**multiplication**" of two GraphBLAS vectors.
- Performs the implied or explicit "multiply" operator (**`mult_op`**) on the **intersection** of the sparse entries in each input vector, u and v.
- **`emult`** defined as a member function of Vector class (also Matrix class)

```
import pygraphblas as grb
u = grb.Vector.sparse(grb.BOOL, NODES);
v = grb.Vector.sparse(grb.BOOL, NODES);

# using an implicit operator (LAND)
v *= u
w = u * v
w = u.emult(v)
u.emult(v, out=w)
```

In-place element-wise multiplication.

These three compute the same result in w.
The third reuses an existing w.

```
# using an explicit operator (LAND)
w = u.emult(v, mult_op=grb.BOOL.LAND)

# using a context manager
with grb.BOOL.LAND:
    w = u.emult(v)
    u.emult(v, out=w)
```

These three compute the same result in w, too.

Element-wise Addition

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

- Compute the element-wise “**addition**” of two GraphBLAS vectors.
- Performs the implied or explicit “addition” operator (**`add_op`**) on the **union** of the sparse entries in each input vector, u and v.
- **`eadd`** defined as a member function of Vector class (also Matrix class)

```
import pygraphblas as grb
u = grb.Vector.sparse(grb.BOOL, NODES);
v = grb.Vector.sparse(grb.BOOL, NODES);

# using an implicit operator (LOR)
v += u
w = u + v
w = u.eadd(v)
u.eadd(v, out=w)
```

In-place element-wise addition.

These three compute the same result in w.
The third reuses an existing w.

```
# using an explicit operator (LOR)
w = u.eadd(v, mult_op=grb.BOOL.LOR)

# using a context manager
with grb.BOOL.LOR:
    w = u.eadd(v)
    u.eadd(v, out=w)
```

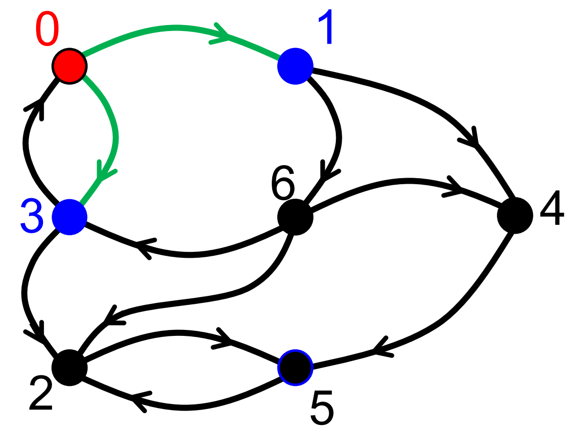
These three compute the same result in w, too.

Exercise 9: Keep track of ‘visited’ nodes

- Modify code from Exercise 8 to compute the visited set as you iterate.

```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
u = grb.Vector.sparse(grb.BOOL, N)
Atrans = A.T
with grb.<type>.<semiring>:
w = A.mxv(u, desc = ??)
v = w.eadd(v)
v += w
A.mxv(u, out=w, desc = ??)
print(A)
draw(A)
```

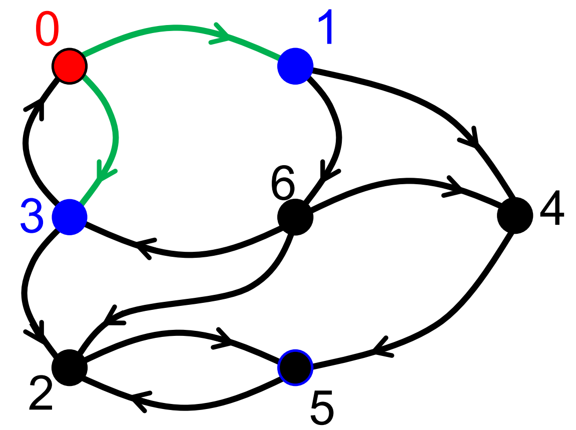
Solution to exercise 9




```
NODES = 7;
v = grb.Vector.sparse(grb.BOOL, NODES);
w = grb.Vector.sparse(grb.BOOL, NODES);
w[0] = True  # 1st wavefront has one node set.

with grb.BOOL.LOR_LAND:
    for i in range(NUM_NODES):
        v.eadd(w, out=v, add_op=grb.BOOL.LOR)        # v += w
        print(v)
        graph.mxv(w, out=w, desc=grb.descriptor.T0) # w = graph.T @ w
        print(w)
```

		i=0	i=1	i=2	i=3	i=4	i=5	i=6
	0	t	t	t	t	t	t	t
	1		t	t	t	t	t	t
	2			t	t	t	t	t
v	3		t	t	t	t	t	t
	4			t	t	t	t	t
	5				t	t	t	t
	6			t	t	t	t	t
	0		t		t		t	
	1	t		t		t		t
	2		t	t	t	t	t	t
w	3	t		t		t		t
	4		t	t	t	t	t	t
	5			t	t	t	t	t
	6		t		t		t	

Solution to exercise 9




```
NODES = 7;
v = grb.Vector.sparse(grb.BOOL, NODES);
w = grb.Vector.sparse(grb.BOOL, NODES);
w[0] = True  # 1st wavefront has one node set.

with grb.BOOL.LOR_LAND:
    for i in range(NUM_NODES):
        v.eadd(w, out=v, add_op=grb.BOOL.LOR)        # v += w
        print(v)
        graph.mxv(w, out=w, desc=grb.descriptor.T0) # w = graph.T @ w
        print(w)
```

What should the exit condition be?

	i=0		i=1								5		i=6	
	0|	t	0|	t								t	0|	t
	1|		1|	t								t	1|	t
	2|		2|		2|	t	2|	t	2|	t	2|	t	2|	t
v	3|		3|	t	3|	t	3|	t	3|	t	3|	t	3|	t
	4|		4|		4|	t	4|	t	4|	t	4|	t	4|	t
	5|		5|		5|		5|	t	5|	t	5|	t	5|	t
	6|		6|		6|	t	6|	t	6|	t	6|	t	6|	t
	0|		0|	t	0|		0|	t	0|		0|	t	0|	
	1|	t	1|		1|	t	1|		1|	t	1|		1|	t
	2|		2|	t	2|	t	2|	t	2|	t	2|	t	2|	t
w	3|	t	3|		3|	t	3|		3|	t	3|		3|	t
	4|		4|	t	4|	t	4|	t	4|	t	4|	t	4|	t
	5|		5|		5|	t	5|	t	5|	t	5|	t	5|	t
	6|		6|	t	6|		6|	t	6|		6|	t	6|	

mxv() revisited

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

- We now need to go back and introduce more notation that will support more efficient graph operations.
- Every GraphBLAS operation that produces a Vector or Matrix result supports an optional ***write mask***.
- Three new descriptor flags can be used to affect mask behavior.

```
import pygraphblas as grb

grb.descriptor.R   # REPLACE flag,     'r'
grb.descriptor.S   # mask STRUCTURE,   's(.)'
grb.descriptor.C   # mask COMPLEMENT,  '¬(.)'

A.mxv(u, out=w, mask=m, desc=grb.descriptor.[R][S][C])
```

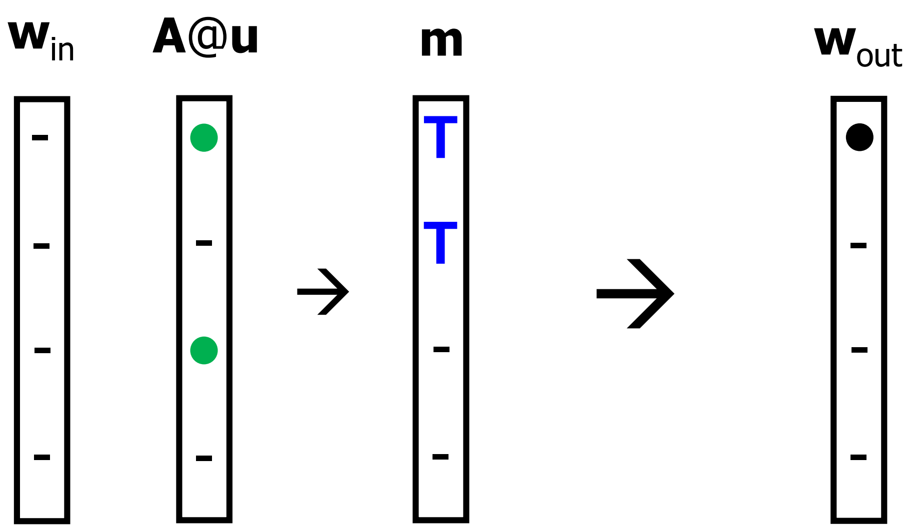
It's time to explain masking and REPLACE in GraphBLAS operations.

Masking

A mask, **m**, is interpreted as a logical ‘stencil’ that controls which elements of the result can be written to the output

- Any location in the mask that evaluates to ‘true’ 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

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

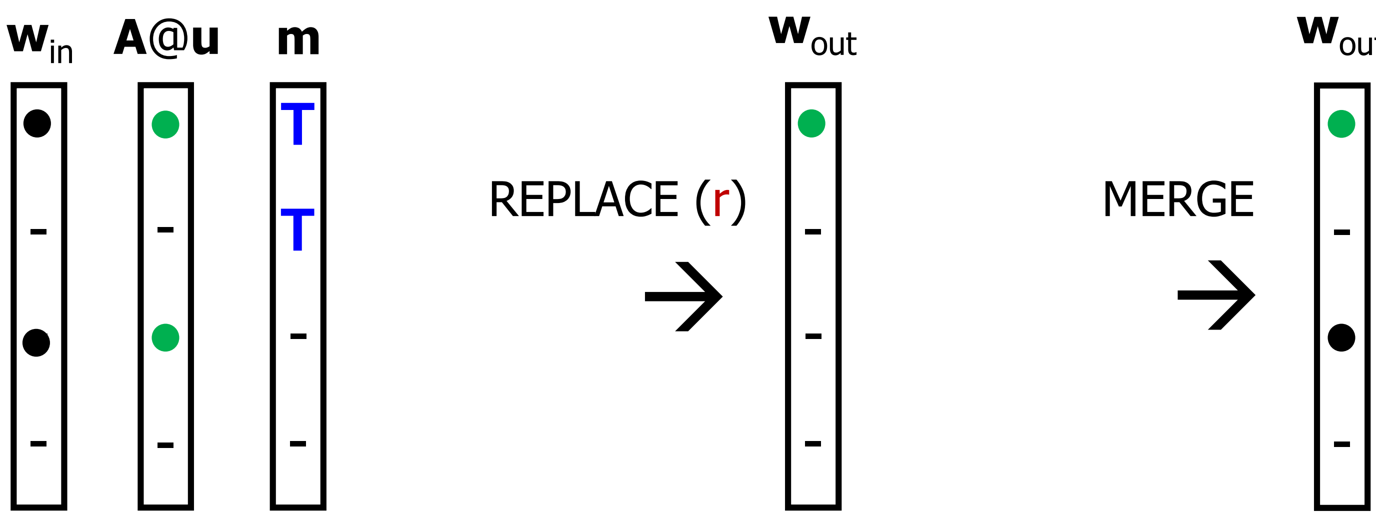



```
A.mxv(u, out=w, mask=m)
```

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 (r): all unwritten locations are cleared (zeroed out).
  - MERGE: all unwritten locations are left unchanged.

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



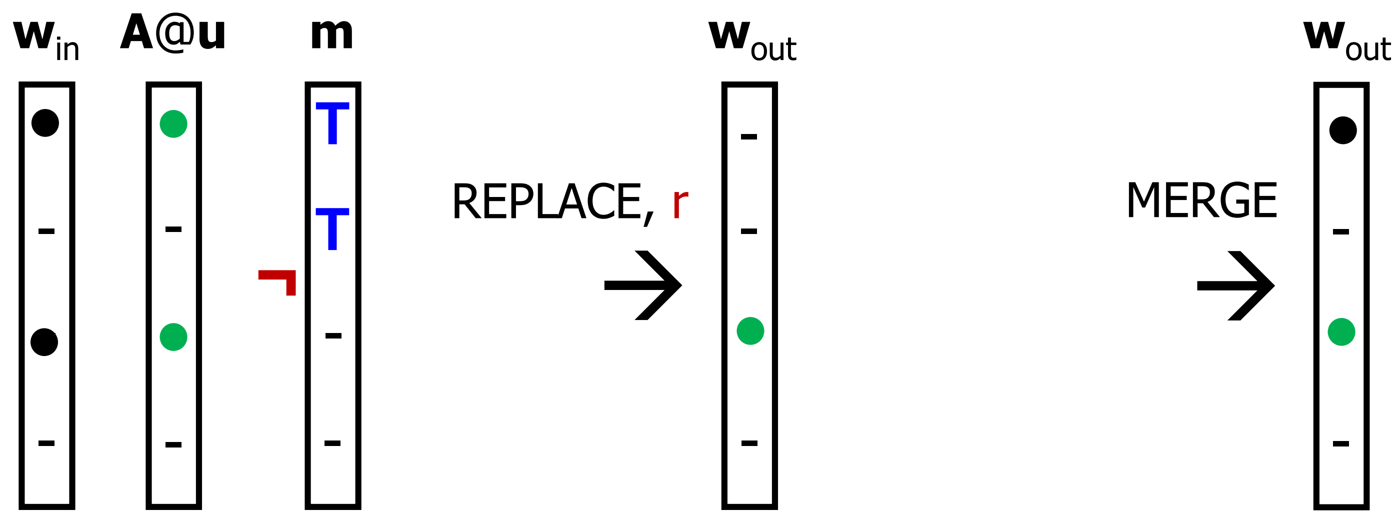
- Behavior defaults to MERGE; otherwise, use a **`descriptor.R`**:

```
A.mxv(u, out=w, mask=m, [desc=grb.descriptor.R])
```

Complement (mask)

- Specified with a descriptor: `grb.descriptor.C`
- Inverts the logic of mask (write enabled on false)

$$\mathbf{w}\langle\neg\mathbf{m}, r\rangle = \mathbf{A} \oplus.\otimes \mathbf{u}$$

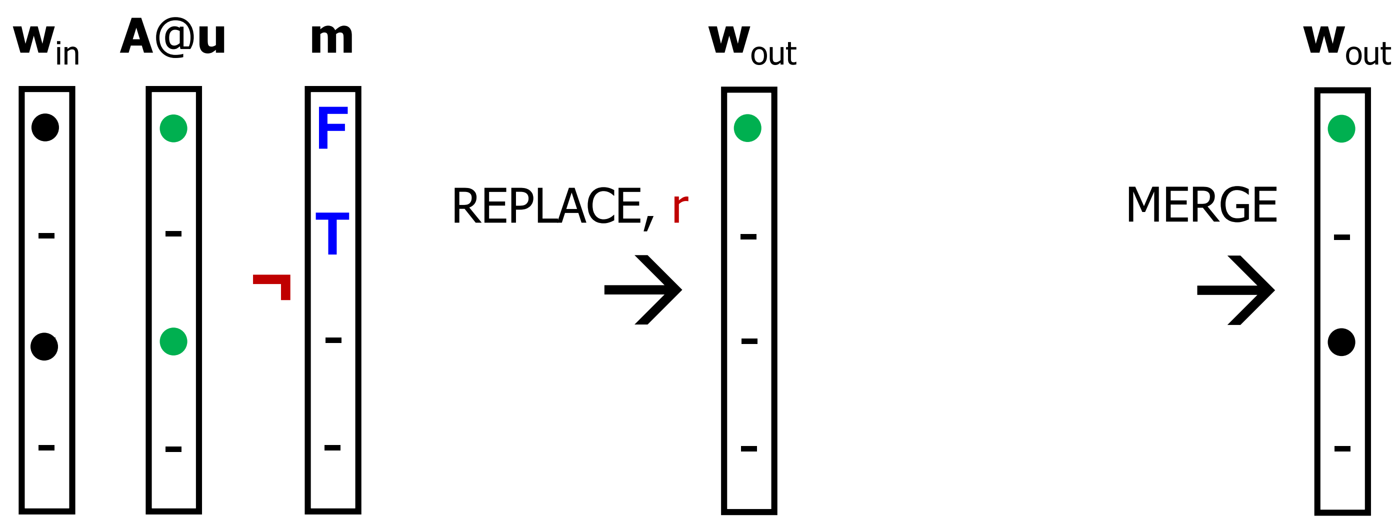



```
A.mxv(u, out=w, mask=m, desc=grb.descriptor.RC)   # REPLACE
A.mxv(u, out=w, mask=m, desc=grb.descriptor.C)    # MERGE
```

Structure only (mask)

- Specified with a descriptor: `grb.descriptor.S`
- Writes are enabled by the pattern of stored values (not the values themselves)

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

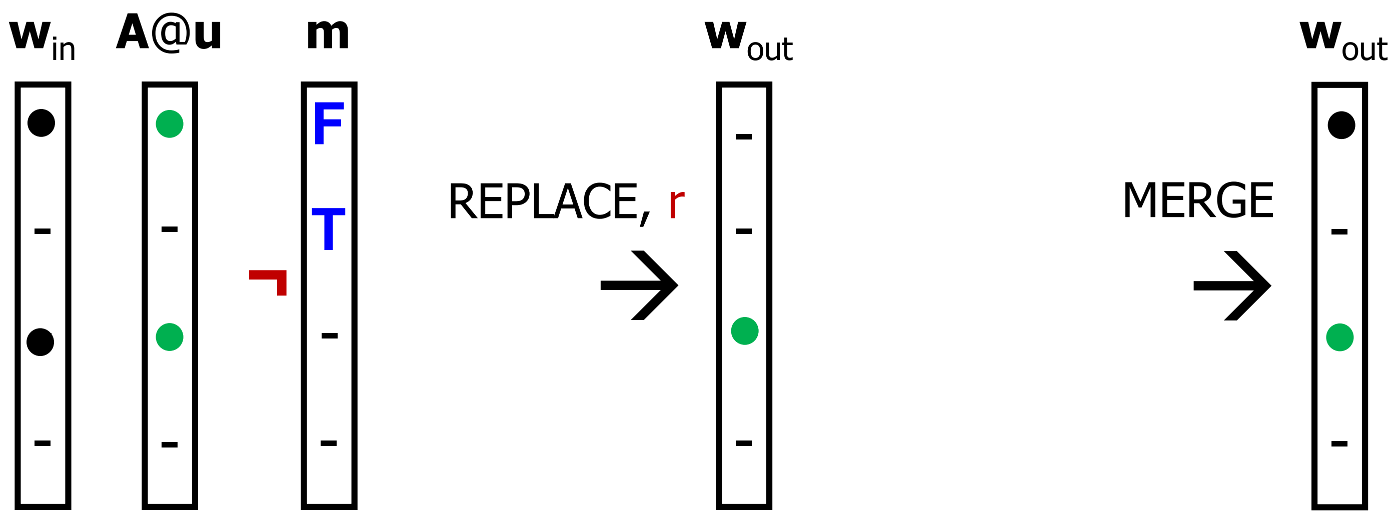



```
A.mxv(u, out=w, mask=m, desc=grb.descriptor.RS)   # REPLACE
A.mxv(u, out=w, mask=m, desc=grb.descriptor.S)    # MERGE
```

Complement the Structure of a mask

- Specified with a descriptor: `grb.descriptor.SC`
- Writes are enabled by the pattern of stored values (not the values themselves)…where values are NOT stored

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




```
A.mxv(u, out=w, mask=m, desc=grb.descriptor.RSC) # REPLACE
A.mxv(u, out=w, mask=m, desc=grb.descriptor.SC)  # MERGE
```

Using Descriptors (summary)

- All of the possible flags for **`grb.descriptor`**:
  - **`R`**: Replace flag (only used when masks are present)
  - **`S`**: Structure of a mask
  - **`C`**: Complement the mask (structure or otherwise)
  - **`T0`**: Transpose the first input (only when it is a Matrix)
  - **`T1`**: Transpose the second input (only when it is a Matrix)

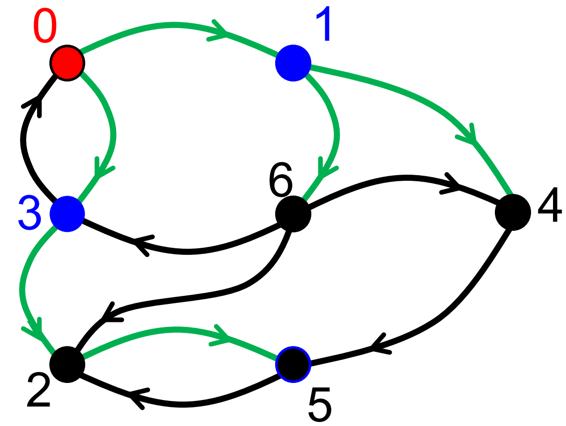
- All 31 combinations of flags are predefined in a set order:
  - **`grb.descriptor.[R][S][C][T0][T1]`**

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 statements, objects and methods from pygraphblas:

```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rowList, columnList, values)
v = grb.Vector.sparse(grb.BOOL, N)
Atrans = A.T
with grb.<type>.<semiring>:
w = A.mxv(u, desc=??)
v = w.eadd(v)
v += w
v.nvals
v.size
A.mxv(u, out=w, desc = ??)
grb.descriptor.[R][S][C][T0][T1]
print(A)
draw(A)
```

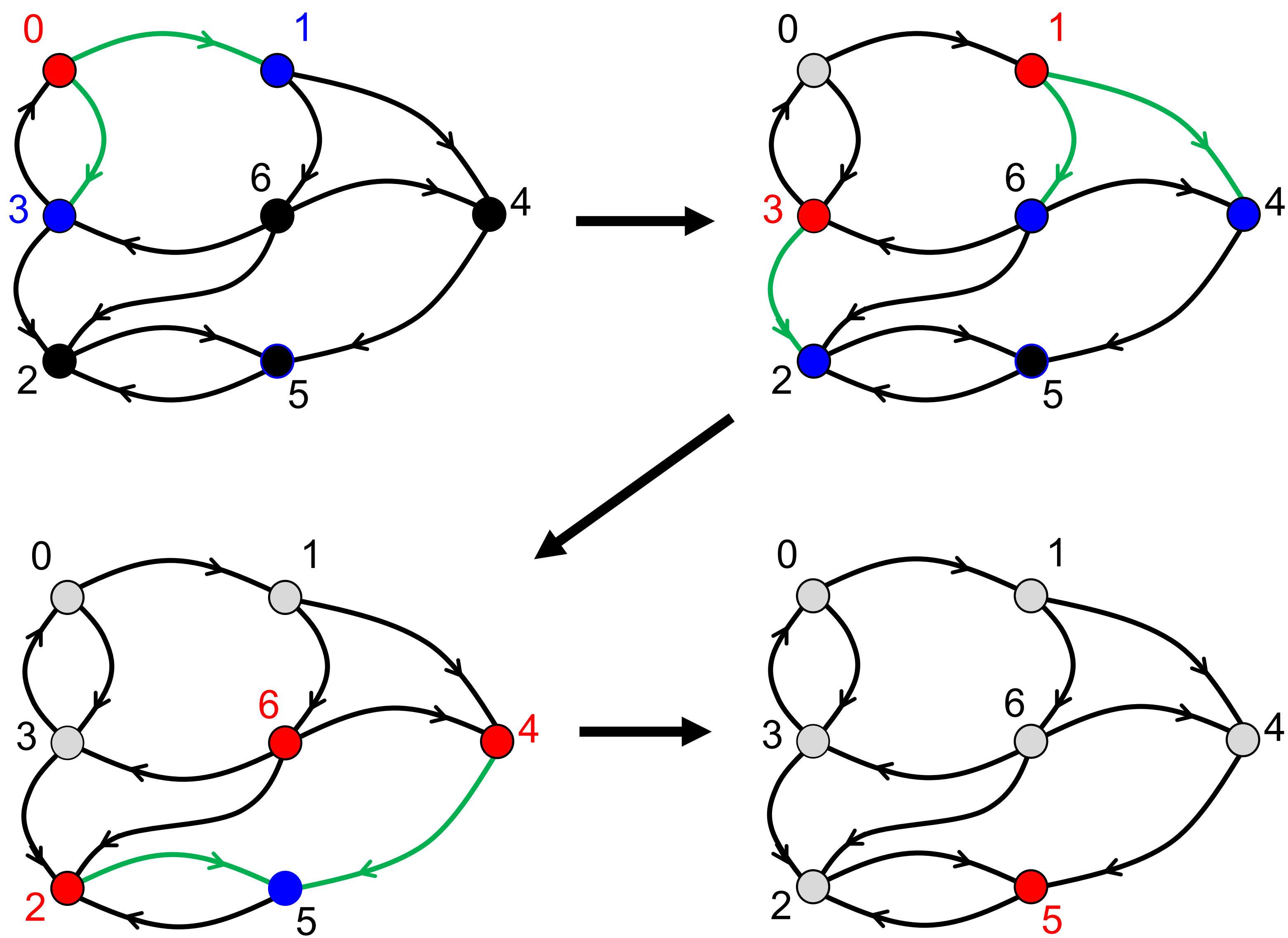
Solution to exercise 10




```
NODES = 7;
v = grb.Vector.sparse(grb.BOOL, NODES);
w = grb.Vector.sparse(grb.BOOL, NODES);
w[0] = True  # 1st wavefront has one node set.

with grb.BOOL.LOR_LAND:
    while w.nvals > 0:
        v.eadd(w, out=v, add_op=grb.BOOL.LOR)  # v += w
        print(v)
        # w<!v, r> = graph.T @ w
        graph.mxv(w, mask=v, out=w, desc=grb.descriptor.RCT0)
        print(w)
```

```
                 i=0        i=1        i=2        i=3

        0|       0| t       0| t       0| t       0| t
        1|       1|         1| t       1| t       1| t
        2|       2|         2|         2| t       2| t
 v      3|       3|         3| t       3| t       3| t
        4|       4|         4|         4| t       4| t
        5|       5|         5|         5|         5| t
        6|       6|         6|         6| t       6| t

        0| t     0|         0|         0|         0|
        1|       1| t       1|         1|         1|
        2|       2|         2| t       2|         2|
 w      3|       3| t       3|         3|         3|
        4|       4|         4| t       4|         4|
        5|       5|         5|         5| t       5|
        6|       6|         6| t       6|         6|
```

Breadth-First Traversal



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

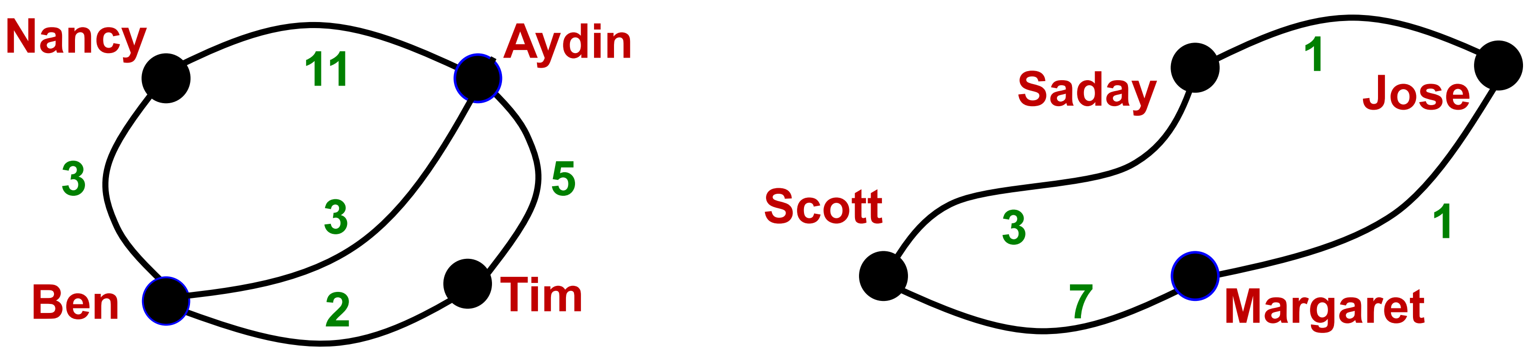
Outline

- Graphs and Linear Algebra
- The GraphBLAS API and Adjacency Matrices
- GraphBLAS Operations
- Pygraphblas and modifying the behavior of operations.
- Graph Algorithms expressed with GraphBLAS
  - 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.

  - For this exercise, the graph will consist of undirected edges
  - Note: applying this to directed graphs by converting to undirected is called "weakly connected components."



assign(), et al.

- There are ***several*** variants of assign
  - Standard vector assignment: `w.assign(u, index=I)`
  - Standard matrix assignment: `C.assign_matrix(A, rindex=I, cindex=J)`

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

  - Assign a vector to the elements of column c_j of a matrix: `C.assign_col(..)`
  - Assign a vector to the elements of row r_i of a matrix: `C.assign_row(..)`

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

  - Assign a constant to a subset of a vector: `w.assign_scalar(..)`
  - Assign a constant to a subset of a matrix: `C.assign_scalar(..)`

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

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

Vector.assign_scalar()

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

```
def assign_scalar(self, value, index = None, mask = None, ...)
```

```
w = grb.Vector.sparse(grb.INT32, 5)
```

- Assign a constant to a list of indices:

```
w.assign_scalar(2, [0,3,4])
```

- Assign a constant to a subset of the output with a mask:

```
m = grb.Vector.from_lists([0,3,4],[True]*3,size=5)
w.assign_scalar(2, mask=m)
```

- Alternative form using the masking:

```
w[m] = 2
```

```
      0| 2
      1|
w =   2|
      3| 2
      4| 2
```

Vector.assign_scalar()

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

```
def assign_scalar(self, value, index = None, mask = None, ...)
```

```
w = grb.Vector.sparse(grb.INT32, 5)
```

- Assign a constant to a list of indices:

```
w.assign_scalar(2, [0,3,4])
```

```
    0| 2
    1|
w = 2|
    3| 2
    4| 2
```

- Assign a constant to a subset of the output with a mask:

```
m = grb.Vector.from_lists([0,3,4],[True]*3,size=5)
w.assign_scalar(2, mask=m)
```

- Alternative form using the masking:

```
w[m] = 2
```

```
    0|
    1| 2
w = 2| 2
    3| 2
    4|
```

- Colon notation is also supported: w[begin:end]:

```
w[1:4] = 2
```

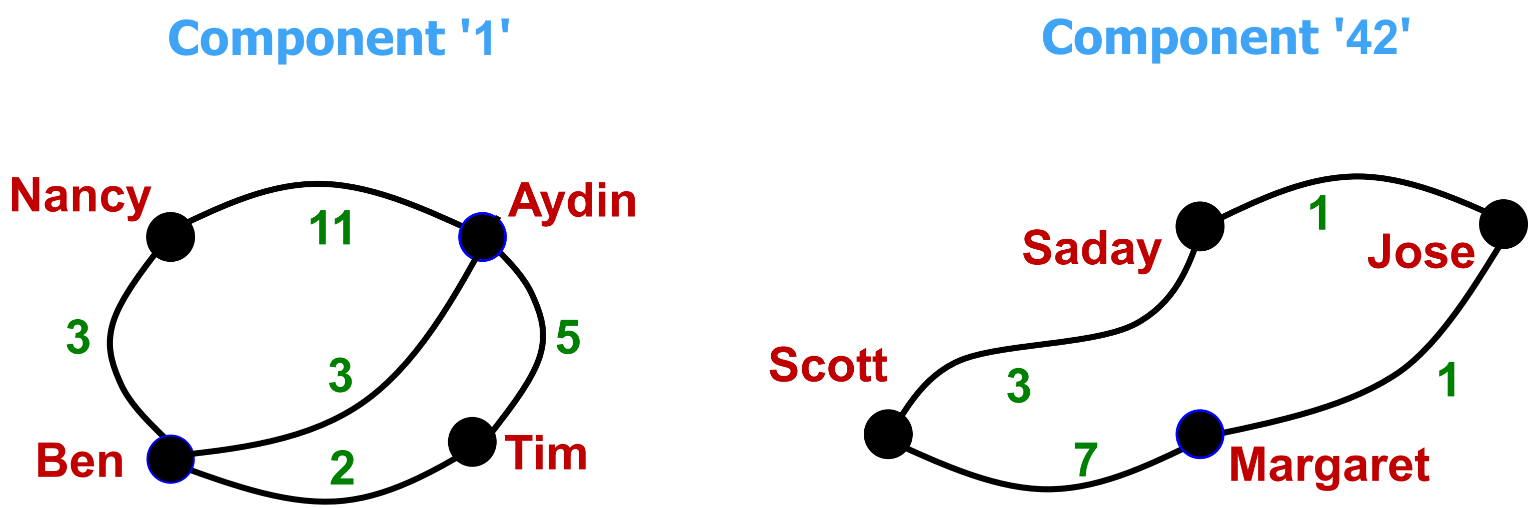
```
    0| 2
    1| 2
w = 2| 2
    3| 2
    4| 2
```

- Equivalent ways to fill a Vector (index=None):

```
w[:] = 2
w.assign_scalar(2)
```

Our Connected Components plan

- Strategy:
  - Create a new vector and 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:

```
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 =
      0  1  2  3  4  5  6  7  8
  0|     1        2           1|  0
  1|  1                       3|  1
  2|                    4      |  2
  3|                 1     2   |  3
  4|  2                       5|  4
  5|           1           2   |  5
  6|        4                  |  6
  7|           2     2         |  7
  8|  1  3        5            |  8
      0  1  2  3  4  5  6  7  8
```

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 =
         0   1   2   3   4   5   6   7   8
    0|       1           2               1|   0
    1|   1                               3|   1
    2|                           4        |   2
    3|                       1       2    |   3
    4|   2                               5|   4
    5|               1               2    |   5
    6|           4                        |   6
    7|               2       2            |   7
    8|   1   3           5                |   8
         0   1   2   3   4   5   6   7   8
```

How many components are there?

Exercise 11: Connected Components

- Wrap the code from Exercise 10 in a function called BFS:

```
def BFS(graph, src_node): # Adjacency Matrix, vertex ID
  ...
  return v                # Boolean Vector with reachable nodes set to True
```

- Call BFS 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.
  - Iterate over the visited list finding unreached nodes to start a BFS
  - Use the following undirected, weighted graph, **with multiple components**:

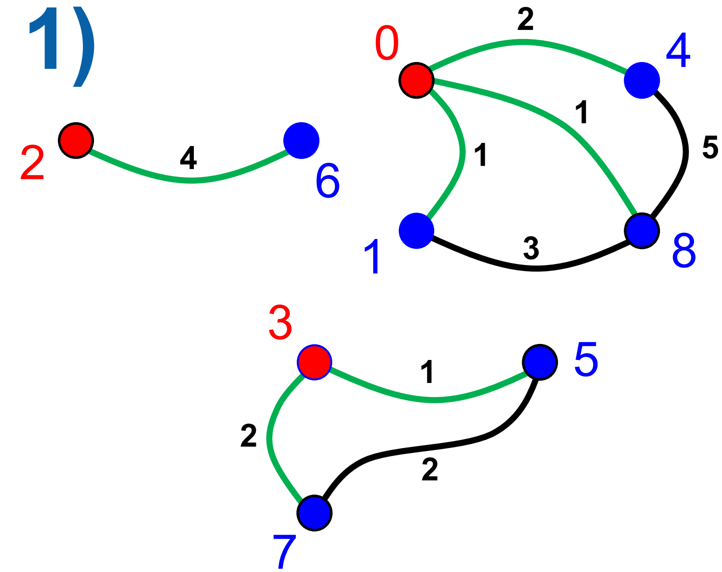
```
row_ind = [0, 0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7, 8, 8, 8]
col_ind = [1, 4, 8, 0, 8, 6, 5, 7, 0, 8, 3, 7, 2, 3, 5, 0, 1, 4]
values  = [1, 2, 1, 1, 3, 4, 1, 2, 2, 5, 1, 2, 4, 2, 2, 1, 3, 5]
```

- Challenge: use `Vector.assign_scalar()` to assign component IDs

```
import pygraphblas as grb
from pygraphblas.gviz import draw_graph as draw
A = grb.Matrix.from_lists(rInd, cInd, vals)
v = grb.Vector.sparse(grb.BOOL, N)
v.assign_scalar(val, index=None, mask=None)
v.get(index)
w = A.mxv(u, desc=??)
v = w.eadd(v), v += w
```

```
v.nvals                       A.nvals
v.size                        A.nrows
Atrans = A.T
A.mxv(u, out=w, desc = ??)
grb.descriptor.[R][S][C][T0][T1]
with grb.<type>.<semiring>:
print(A)
draw(A)
```

Solution to exercise 11 (part 1)

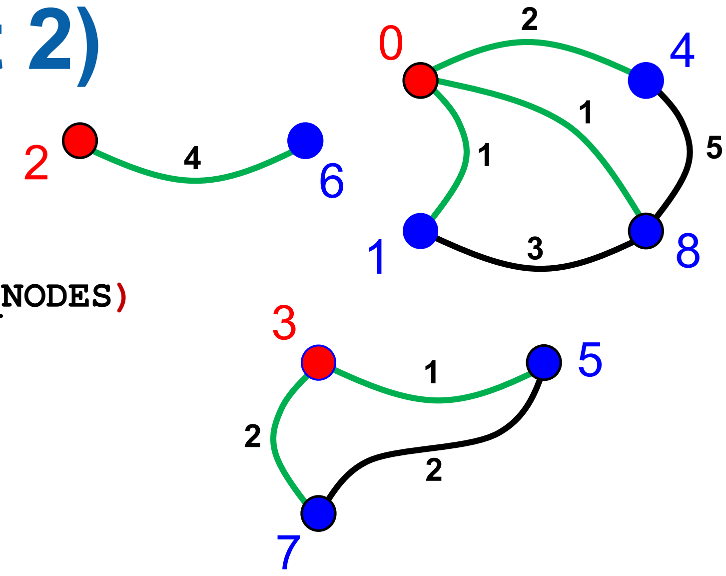



```
def BFS(graph, src_node):
    NODES = graph.nrows;
    v = grb.Vector.sparse(grb.BOOL, NODES);
    w = grb.Vector.sparse(grb.BOOL, NODES);
    w[src_node] = True  # 1st wavefront

    with grb.BOOL.LOR_LAND:
        while w.nvals > 0:
            #v.eadd(w, out=v, add_op=grb.BOOL.LOR) # v += w
            v.assign_scalar(True, mask=w)          # v[w] = True

            # w<!v, r> = graph.T @ w
            graph.mxv(w, mask=v, out=w, desc=grb.descriptor.RCT0)

    return v
```

Mostly the same as Exercise 10:

- Need to get the number of nodes from matrix properties, i.e., nrows
- This illustrates another way to accumulate the visited list using assign

Solution to exercise 11 (part 2)

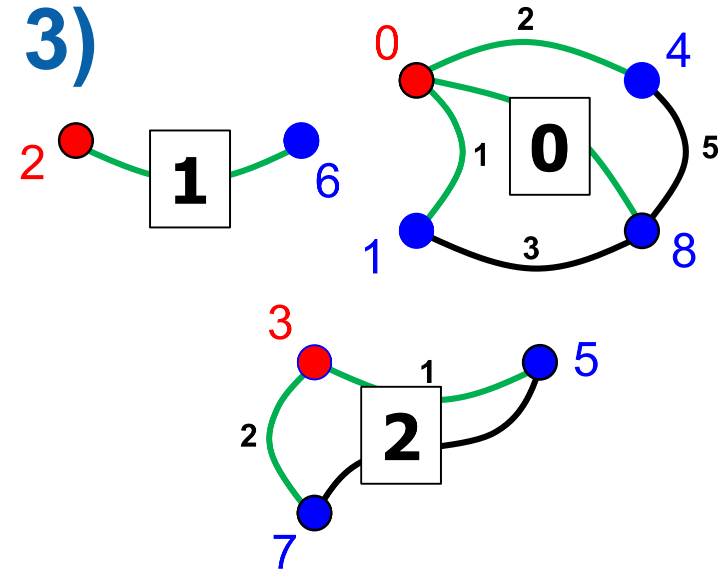



```
def CC(graph):
    NUM_NODES = graph.nrows
    cc_ids    = grb.Vector.sparse(grb.UINT64, NUM_NODES)
    num_ccs   = 0

    for src_node in range(NUM_NODES):
        if cc_ids.get(src_node) is None:
            print("Processing node", src_node)

            # find all nodes reachable from src_node
            visited = BFS(graph, src_node)

            cc_ids.assign_scalar(num_ccs, mask=visited)  # cc_ids[visited]=num_ccs
            num_ccs += 1

    return num_ccs, cc_ids
```

Notes:

- If **`cc_ids.get(index)`** returns **`None`**, there is no stored value at that location (it has not been visited yet).
- Using **`assign_scalar`** to assign component IDs to visited vertices
- Not shown: opportunity for early exit when cc_ids.nvals == NUM_NODES

Solution to exercise 11 (part 3)




```
def BFS(graph, src_node):
  ...
  return v

def CC(graph):
  ...
  return num_ccs, cc_ids

row_ind = [0, 0, 0, 1, 1, 2, 3, ...]
col_ind = [1, 4, 8, 0, 8, 6, 5, ...]
values  = [1, 2, 1, 1, 3, 4, 1, ...]
graph = grb.Matrix.from_lists(row_ind, col_ind, values)

num_ccs, cc_ids = CC(graph)

print("\nFound", num_ccs, "components.")
print(cc_ids)

draw(graph)
draw(graph, label_vector = cc_ids)
```

Output:

```
Processing node 0
Processing node 2
Processing node 3

Found 3 components.
0|  0
1|  0
2|  1
3|  2
4|  0
5|  2
6|  1
7|  2
8|  0
```

Putting it all together…

- Copying CC algorithm to AnalyzeGraph notebook…

```
.
.
.
*** Step 3a: Running Tutorial connected components algorithm.
Largest component #0 (size = 822)
*** Step 3a: Elapsed time: 0.0222051 sec*
Found 246 components
```

- Check out the LAGraph repository for *significantly* more efficient algorithms** written in C for the GraphBLAS (coming soon to python)
  - https://github.com/GraphBLAS/LAGraph

*On one core of i9-9900 @ 3.10GHz

**Azad, Buluç. "LACC: a linear-algebraic algorithm for finding connected components in distributed memory" (IPDPS 2019).

**Zhang, Azad, Hu. "FastSV: FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence" (SIAM PP20).

The GraphBLAS Operations

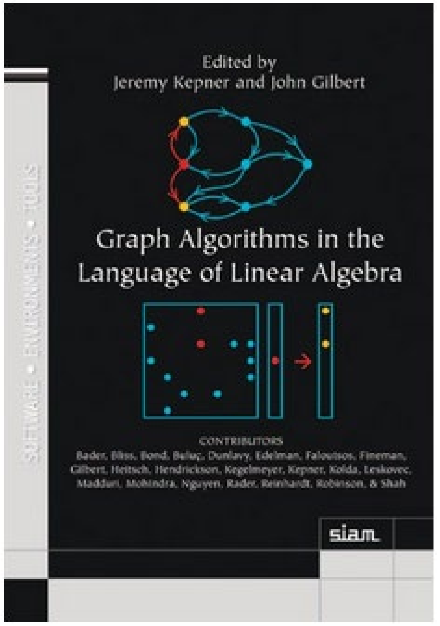
Operation Name	Mathematical Notation				
mxm	$\mathbf{C}\langle\mathbf{M},z\rangle$	$=$	$\mathbf{C}$	$\odot$	$\mathbf{A}\oplus.\otimes\mathbf{B}$
mxv	$\mathbf{w}\langle\mathbf{m},z\rangle$	$=$	$\mathbf{w}$	$\odot$	$\mathbf{A}\oplus.\otimes\mathbf{u}$
vxm	$\mathbf{w}^T\langle\mathbf{m}^T,z\rangle$	$=$	$\mathbf{w}^T$	$\odot$	$\mathbf{u}^T\oplus.\otimes\mathbf{A}$
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	$\mathbf{C}\langle\mathbf{M},z\rangle$	$=$	$\mathbf{C}$	$\odot$	$\mathbf{A}(\mathbf{i},\mathbf{j})$
	$\mathbf{w}\langle\mathbf{m},z\rangle$	$=$	$\mathbf{w}$	$\odot$	$\mathbf{u}(\mathbf{i})$
assign	$\mathbf{C}\langle\mathbf{M},z\rangle(\mathbf{i},\mathbf{j})$	$=$	$\mathbf{C}(\mathbf{i},\mathbf{j})$	$\odot$	$\mathbf{A}$
	$\mathbf{w}\langle\mathbf{m},z\rangle(\mathbf{i})$	$=$	$\mathbf{w}(\mathbf{i})$	$\odot$	$\mathbf{u}$

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 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 →



Appendices

- MxM: the low-level details of the GraphBLAS operations
- Common patterns for algorithmic reasoning
- Challenge Problems:
  - Some key algorithms with the GraphBLAS
- Reference materials

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

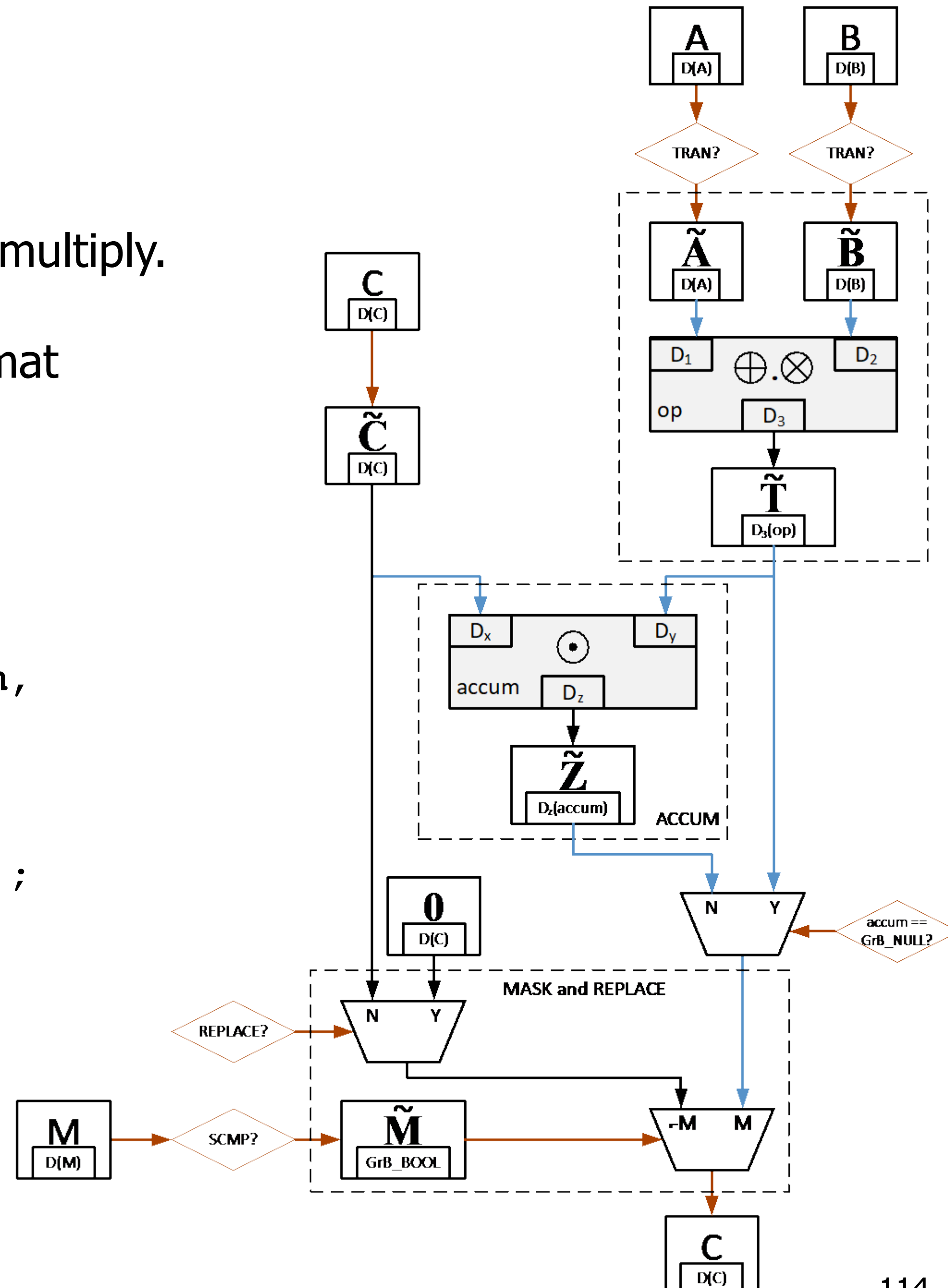
Form input operands and mask based on descriptor	C, B, M, A ← A^T
Test the domains and sizes for consistency.	int32, dims match
Carry out the indicated operation	T ← A *.+ B, Z ← C + T
Apply the write-mask to select output values	Z ← Z ∩ M
Replace mode: delete elements in output object and replace with 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.

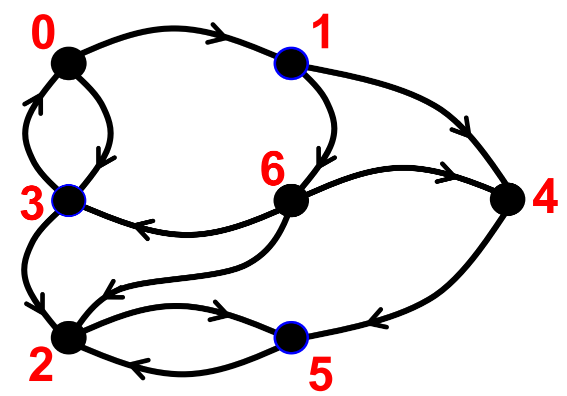
All the operations follow this basic format

```
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);
```



Exercise: Matrix Matrix Multiplication

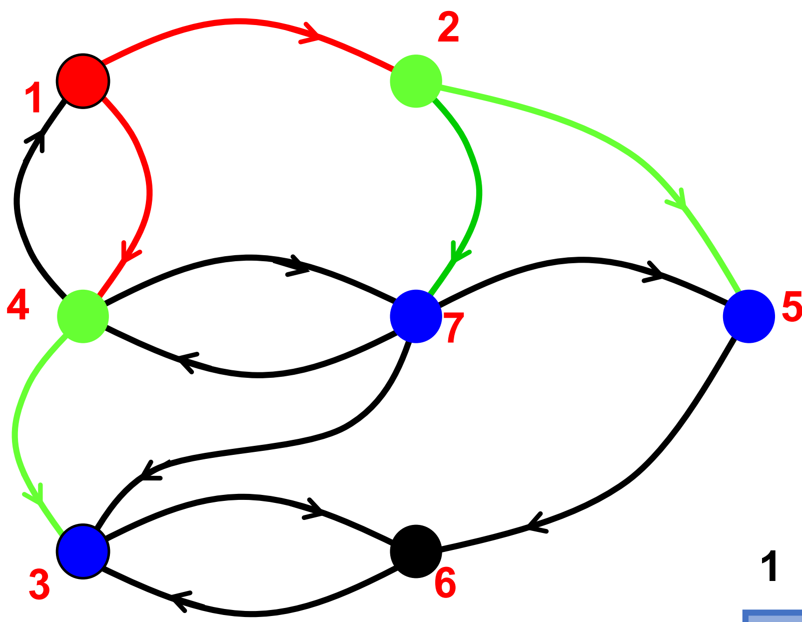
- Multiply the adjacency matrix from our “logo graph” by itself.
- Print resulting matrix and interpret the result
- Hint: Do the multiply again and compare results. Do you see the pattern?



Appendices

- MxM: the low-level details of the GraphBLAS operations
- Common patterns for algorithmic reasoning
- Challenge Problems:
  - Some key algorithms with the GraphBLAS
- Reference materials

SpMSpV: Sparse-Matrix/Sparse-Vector Multiplication



SpMSpV: single-source graph traversal

Used in BFS, CC, matching, ordering, etc.

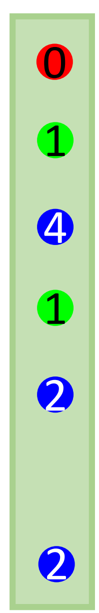
`y = A.mxv(x, mask=p, desc=descriptor.T0)`

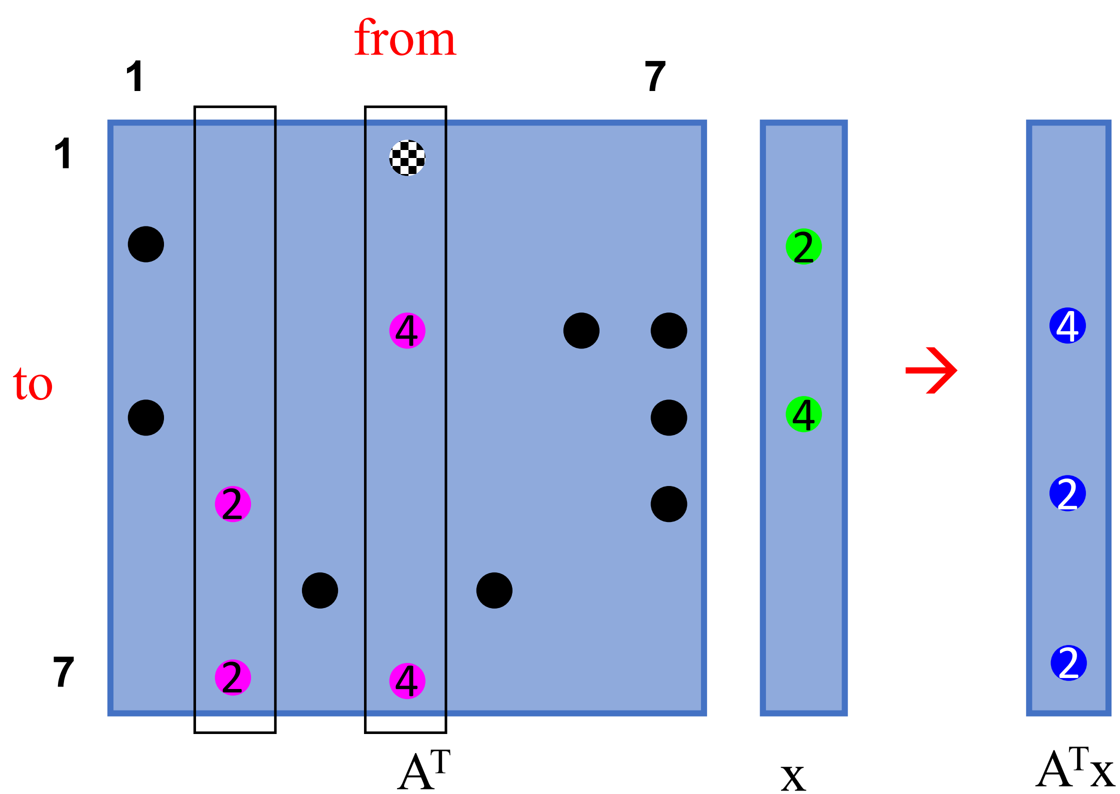
A: sparse adjacency matrix

x: sparse input vector (previous frontier)

p: mask (already discovered vertices)

parents (p):





SpMSpM: Sparse-Matrix/Sparse-Matrix Multiplication

SpGEMM: multi-source graph traversal

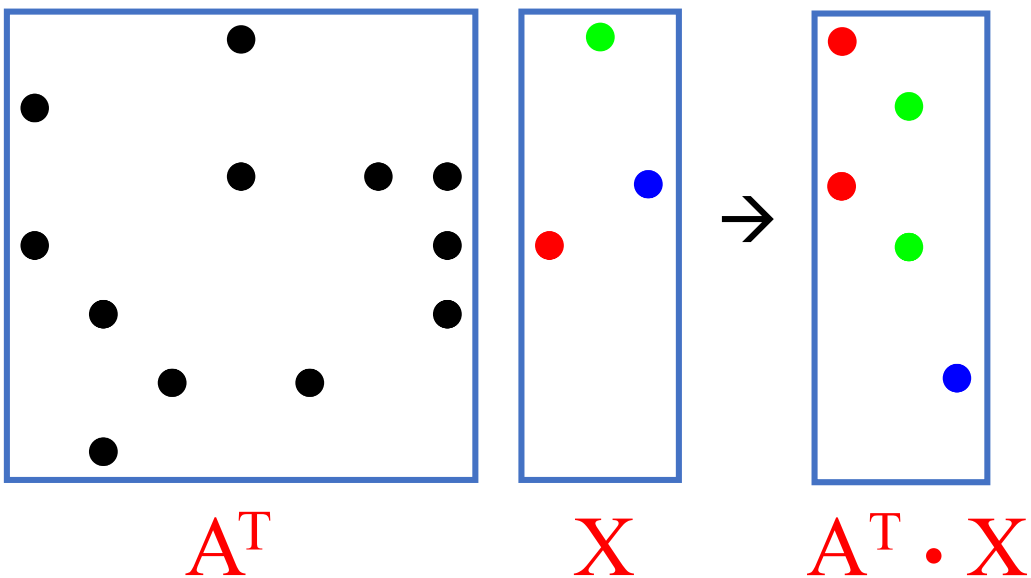
Ex: multi-source BFS, betweenness centrality, triangle counting*, Markov clustering*

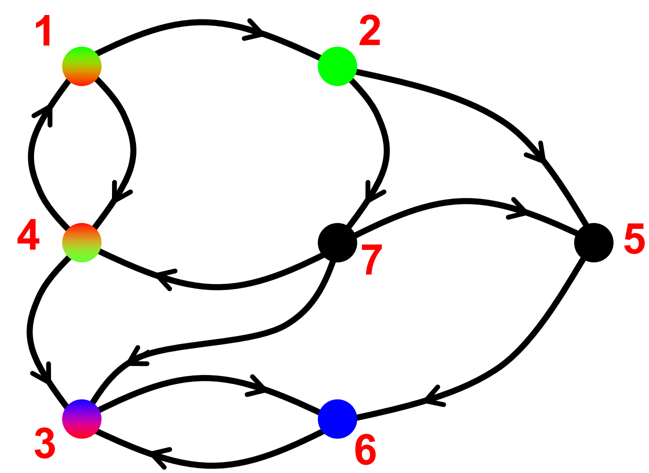
Y=A.mxm(X, mask=P, desc=descriptor.T0)

A: sparse adjacency matrix

X: sparse input matrix (previous frontier), n-by-b where b is the #sources

P: mask (already discovered vertices), multi-vector version of p from previous slide





*: shown in later slides

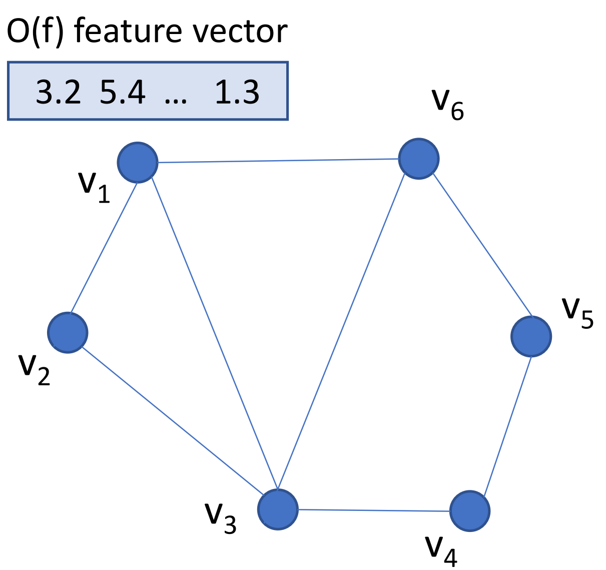
SpMM: Sparse-Matrix/dense-Matrix Multiplication

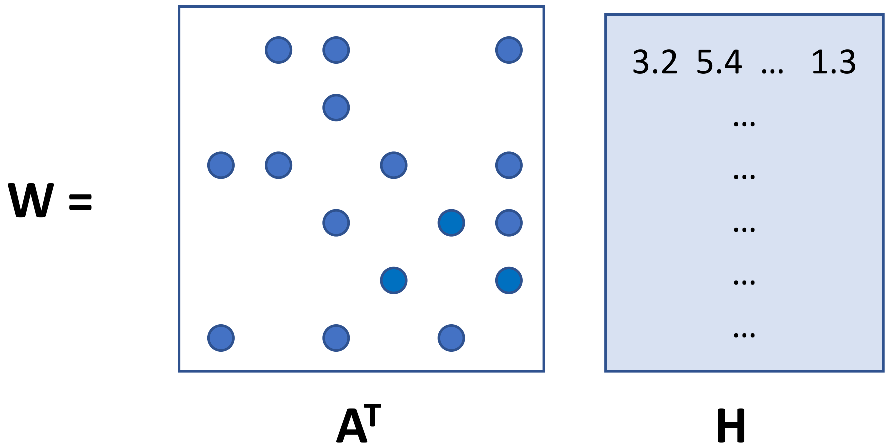
SpMM: feature aggregation from neighbors
Used in Graph neural networks, graph embedding, etc.

W=A.mxm(H, desc=descriptor.T0)

A: sparse adjacency matrix, n-by-n
H: input dense matrix, n-by-f where f << n is the feature dimension
W: output dense matrix, new features

We call this dense matrix, H, a "tall skinny" Matrix

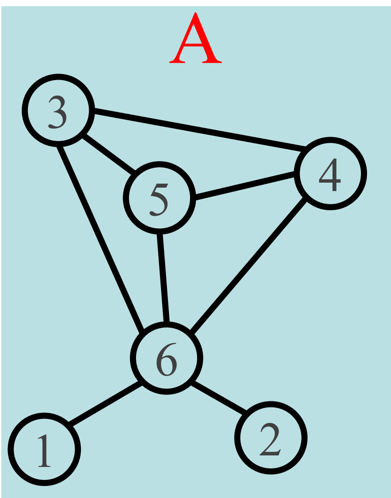




SpMSpM Example: Triangle Counting

Triangle counting is also multi-source(in fact, all sources) traversal.
It just stops after one traversal iteration only, discovering all wedges

B=L.mxm(U)

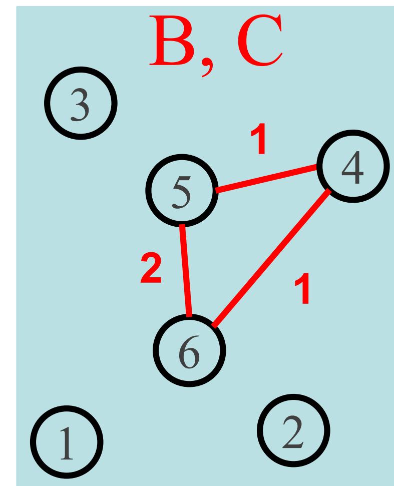


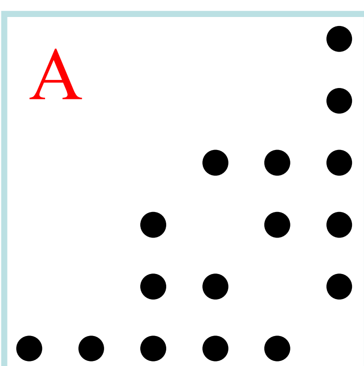
A = L + U (hi->lo + lo->hi)

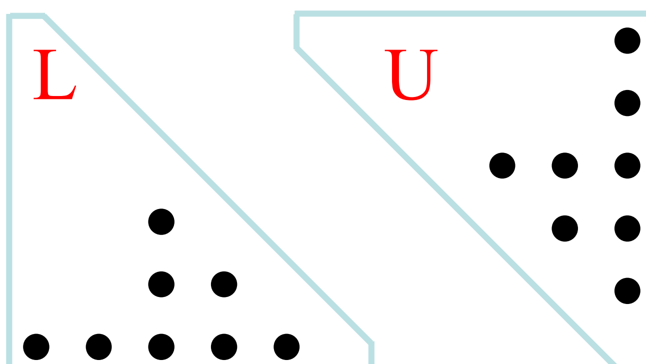
L × U = B (wedge, low hinge)

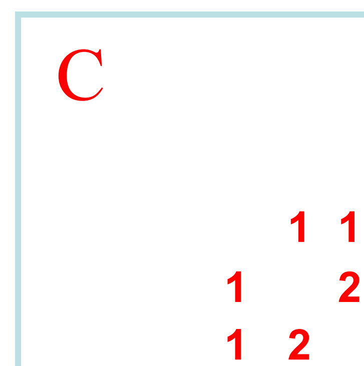
A ^ B = C (closed wedge)

sum(C)/2 = **4 triangles**







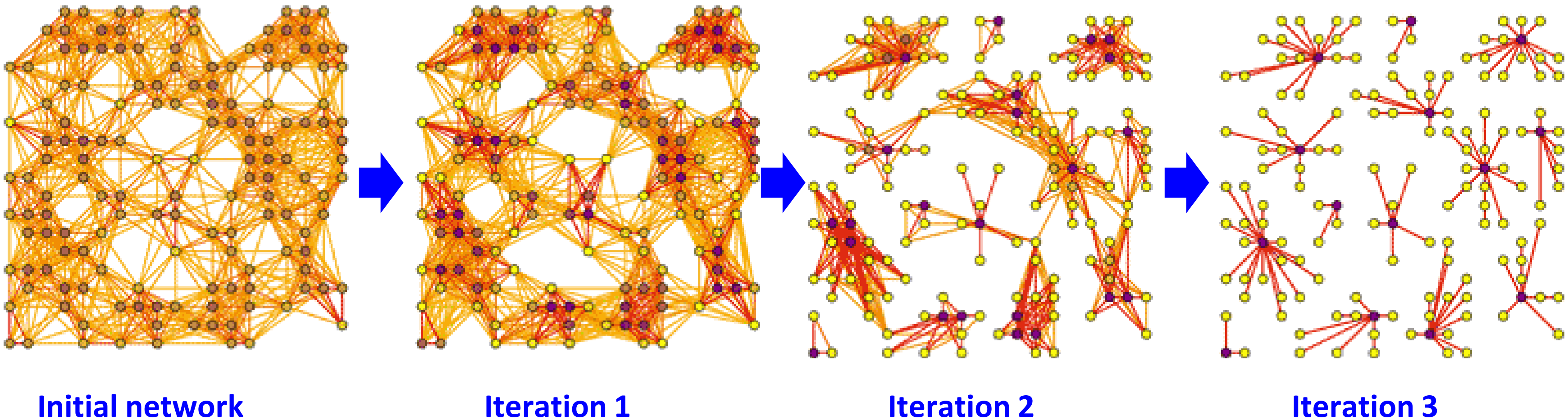


SpMSpM Example: Triangle Counting

Markov clustering is also multi-source (in fact, all sources) traversal.
It alternates between SpGEMM and element-wise or column-wise pruning

C=A.mxm(A, desc=descriptor.T0T1)

A: sparse normalized adjacency matrix
C: denser (but still sparse) pre-pruned matrix for next iteration



At each iteration:
Step 1 (Expansion): Squaring the matrix while pruning (a) small entries, (b) denser columns
Naïve implementation: sparse matrix-matrix product (SpGEMM), followed by column-wise top-K selection and column-wise pruning
Step 2 (Inflation) : taking powers entry-wise

Appendices

- MxM: the low-level details of the GraphBLAS operations
- Common patterns for algorithmic reasoning
- Challenge Problems:
  - Some key algorithms with the GraphBLAS
- Reference materials

Challenge problems

- Triangle counting
- Direction optimizing Breadth first search
- Betweenness Centrality

Notation

operation/method	description	notation

`mxm`
`vxm`
`mxv`
	matrix-matrix multiplication
	vector-matrix multiplication
	matrix-vector multiplication
		$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A} \oplus.\otimes \mathbf{B}$
		$\mathbf{w}^{\mathsf{T}}\langle\mathbf{m}^{\mathsf{T}}\rangle\odot=\mathbf{u}^{\mathsf{T}} \oplus.\otimes \mathbf{A}$
		$\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{A} \oplus.\otimes \mathbf{u}$

`eWiseAdd`	element-wise addition using operator **op** on elements in the set union of structures of $\mathbf{A}/\mathbf{B}$ and **u/v**	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\ \mathbf{op}_{\cup}\ \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\ \mathbf{op}_{\cup}\ \mathbf{v}$
`eWiseMult`	element-wise multiplication using operator **op** on elements in the set intersection of structures of $\mathbf{A}/\mathbf{B}$ and **u/v**	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\ \mathbf{op}_{\cap}\ \mathbf{B}$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\ \mathbf{op}_{\cap}\ \mathbf{v}$
`extract`	extract submatrix from matrix $\mathbf{A}$ using indices $\boldsymbol{i}$ and indices $\boldsymbol{j}$ extract the jth column vector from matrix $\mathbf{A}$ extract subvector from $\mathbf{u}$ using indices $\boldsymbol{i}$	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}(\boldsymbol{i},\boldsymbol{j})$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{A}(:,j)$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}(\boldsymbol{i})$
`assign`	assign matrix to submatrix with mask for $\mathbf{C}$ assign scalar to submatrix with mask for $\mathbf{C}$ assign vector to subvector with mask for $\mathbf{w}$ assign scalar to subvector with mask for $\mathbf{w}$	$\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot=\mathbf{A}$ $\mathbf{C}\langle\mathbf{M}\rangle(\boldsymbol{i},\boldsymbol{j})\odot=s$ $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot=\mathbf{u}$ $\mathbf{w}\langle\mathbf{m}\rangle(\boldsymbol{i})\odot=s$
`subassign (GxB)`	assign matrix to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$ assign scalar to submatrix with submask for $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})$ assign vector to subvector with submask for $\mathbf{w}(\boldsymbol{i})$ assign scalar to subvector with submask for $\mathbf{w}(\boldsymbol{i})$	$\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot=\mathbf{A}$ $\mathbf{C}(\boldsymbol{i},\boldsymbol{j})\langle\mathbf{M}\rangle\odot=s$ $\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot=\mathbf{u}$ $\mathbf{w}(\boldsymbol{i})\langle\mathbf{m}\rangle\odot=s$
`apply`	apply unary operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot=f(\mathbf{A},k)$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=f(\mathbf{u},k)$
`select`	apply select operator f with optional thunk k	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}\langle f(\mathbf{A},k)\rangle$ $\mathbf{w}\langle\mathbf{m}\rangle\odot=\mathbf{u}\langle f(\mathbf{u},k)\rangle$
`reduce`	row-wise reduce matrix to column vector reduce matrix to scalar reduce vector to scalar	$\mathbf{w}\langle\mathbf{m}\rangle\odot=[\oplus_j\ \mathbf{A}(:,j)]$ $s\odot=[\oplus_{i,j}\ \mathbf{A}(i,j)]$ $s\odot=[\oplus_i\ \mathbf{u}(i)]$
`transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle\odot=\mathbf{A}^{\mathsf{T}}$
`dup`	duplicate matrix duplicate vector	$\mathbf{C} \leftarrow \mathbf{A}$ $\mathbf{w} \leftarrow \mathbf{u}$
`build`	matrix from tuples vector from tuples	$\mathbf{C} \leftarrow \{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\}$ $\mathbf{w} \leftarrow \{\boldsymbol{i},\boldsymbol{x}\}$
`extractTuples`	extract index arrays $(\boldsymbol{i},\boldsymbol{j})$ and value arrays $(\boldsymbol{x})$	$\{\boldsymbol{i},\boldsymbol{j},\boldsymbol{x}\} \leftarrow \mathbf{A}$ $\{\boldsymbol{i},\boldsymbol{x}\} \leftarrow \mathbf{u}$
`extractElement`	extract element to scalar	$s = \mathbf{A}(i,j)$ $s = \mathbf{u}(i)$
`setElement`	set element	$\mathbf{C}(i,j) = s$ $\mathbf{w}(i) = s$

Triangle Counting

Algorithm 6: Triangle counting.

Data: $\mathbf{A} \in \mathbb{B}^{n \times n}$

Result: $t \in$ `UINT64`

Function *TriangleCount*

  sample the *mean* and *median* degree of $\mathbf{A}$

  if $mean > 4 \times median$ **then**

    $\mathbf{p}$ = permutation to sort degree, ascending order

    $\mathbf{A} = \mathbf{A}(\mathbf{p}, \mathbf{p})$

  $\mathbf{L} = \mathrm{tril}(\mathbf{A})$

  $\mathbf{U} = \mathrm{triu}(\mathbf{A})$

  $\mathbf{C}\langle s(\mathbf{L})\rangle = \mathbf{L}$ plus.pair $\mathbf{U}^{\mathsf{T}}$

  $t = [+_{ij}\, \mathbf{C}(i, j)]$

Breadth First Search

Algorithm 2: Direction-Optimizing Parent BFS.

Input: $\mathbf{A}$, $\mathbf{A}^{\mathsf{T}}$, $startVertex$

Function $DirectionOptimizingBFS$

  $\mathbf{q}(startVertex) = 0$

  for $level = 1$ **to** $\mathrm{nrows}(\mathbf{A}) - 1$ **do**

    if $Push(\mathbf{A}, \mathbf{q})$ **then**

      $\mathbf{q}^{\mathsf{T}}\langle\neg s(\mathbf{p}^{\mathsf{T}}), \mathrm{r}\rangle = \mathbf{q}^{\mathsf{T}}$ any.secondi $\mathbf{A}$

    else

      $\mathbf{q}\langle\neg s(\mathbf{p}), \mathrm{r}\rangle = \mathbf{A}^{\mathsf{T}}$ any.secondi $\mathbf{q}$

    $\mathbf{p}\langle s(\mathbf{q})\rangle = \mathbf{q}$

    if $\mathrm{nvals}(\mathbf{q}) = 0$ **then**

      return

Betweenness Centrality

Algorithm 3: Betweenness centrality.

Function *BrandesBC*

// $\mathbf{P}(k, j)$ = # paths from kth source to node j

// $\mathbf{F}$: # paths in the current frontier

let: $\mathbf{P} \in \mathbb{Q}_{64}^{ns \times n}$

let: $\mathbf{F} \in \mathbb{Q}_{64}^{ns \times n}$

$\mathbf{P}(1 : k, \mathbf{s}) = 1$

// First frontier:

$\mathbf{F}\langle\neg s(\mathbf{P})\rangle = \mathbf{P}$ plus.first $\mathbf{A}$

// BFS phase:

for $d = 0$ **to** $\mathrm{nrows}(\mathbf{A})$ **do**

  let: $\mathbf{S}[d] \in \mathbb{B}^{ns \times n}$

  $\mathbf{S}[d]\langle s(\mathbf{F})\rangle = 1$ // $\mathbf{S}[d]$ = pattern of $\mathbf{F}$

  $\mathbf{P} \mathrel{+}= \mathbf{F}$

  $\mathbf{F}\langle\neg s(\mathbf{P}), \mathrm{r}\rangle = \mathbf{F}$ plus.first $\mathbf{A}$

  if $\mathrm{nvals}(\mathbf{F}) = 0$ **then**

    break

// Backtrack phase:

let: $\mathbf{B} \in \mathbb{Q}_{64}^{ns \times n}$

$\mathbf{B}(:) = 1.0$

let: $\mathbf{W} \in \mathbb{Q}_{64}^{ns \times n}$

for $i = d - 1$ **downto** 0 **do**

  $\mathbf{W}\langle s(\mathbf{S}[i]), \mathrm{r}\rangle = \mathbf{B}\ \mathrm{div}_{\cap}\ \mathbf{P}$

  $\mathbf{W}\langle s(\mathbf{S}[i-1]), \mathrm{r}\rangle = \mathbf{W}$ plus.first $\mathbf{A}^{\mathsf{T}}$

  $\mathbf{B} \mathrel{+}= \mathbf{W} \times_{\cap} \mathbf{P}$

// $\mathrm{centrality}(j) = \sum_i (\mathbf{B}(i, j) - 1)$

$\mathrm{centrality}(:) = -ns$

$\mathrm{centrality} \mathrel{+}= [+_i\, \mathbf{B}(i, :)]$

Appendices

- MxM: the low-level details of the GraphBLAS operations
- Common patterns for algorithmic reasoning
- Challenge Problems:
  - Some key algorithms with the GraphBLAS
- Reference materials

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.

GraphBLAS predefined operators

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

Identifier	**Domains**	**Description**	
GrB_LOR	bool x bool → bool	$f(x,y) = x \lor y$	Logical OR
GrB_LAND	bool x bool → bool	$f(x,y) = x \land y$	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.