SuiteSparse:GraphBLAS: graph algorithms via sparse matrix operations on semirings

Tim Davis
Texas A&M University

Sept 2017

Sparse Days 2017 at CERFACS

Outline

- Graph algorithms in the language of linear algebra
  - Consider `C=A*B` on a *semiring*
  - Semiring: an add operator, multiply operator, and additive identity
  - Example: with OR-AND: `A` and `B` are adjacency matrices of two graphs
  - `C`: contains edge (i, j) if nodes i and j share any neighbor in common
  - written as `C = A or.and B` or $C = A|.\&B$
- The GraphBLAS Spec: graphblas.org
- SuiteSparse:GraphBLAS implementation and performance

Breadth-first search in pseudo-MATLAB notation

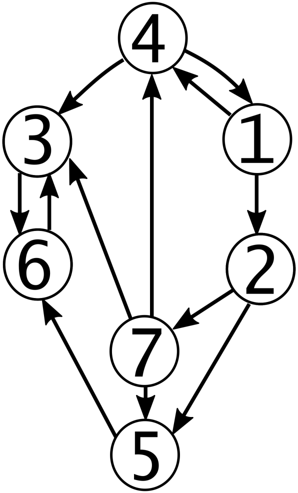
```
v = zeros (1,n) ;         % v(k) is the BFS level (1 for source node)
q = false (1,n) ;         % boolean vector of size n
q (source) = true ;       % q: boolean vector of current level

for level = 1:n
    v (q) = level ;       % set v(i)=level where q(i) is true

    % new q = all unvisited neighbors of current q:
    t = A*q ;             % where '*' is the OR-AND semiring
    q = false (1,n) ;     % clear q of all entries
    q (~v) = t ;          % q (i) = t (i) but only where v(i) is zero

    if (~any (q)) break ;
end
```

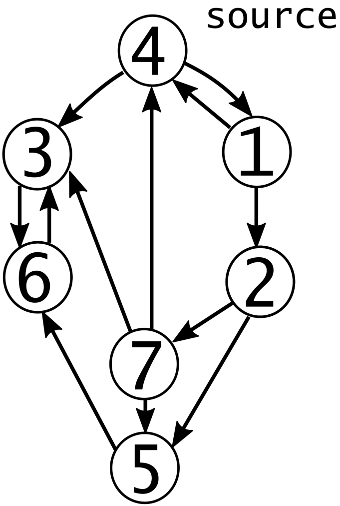
Breadth-first search example



$A(i, j) = 1$ for edge (j, i)
A is binary; shown with integers to illustrate row indices; each column is an adjacency list, and dot (.) is zero:

```
. . . 1 . . .
2 . . . . . .
. . . 3 . 3 3
4 . . . . . 4
. 5 . . . . 5
. . 6 . 6 . .
. 7 . . . . .
```

Breadth-first search: initialization

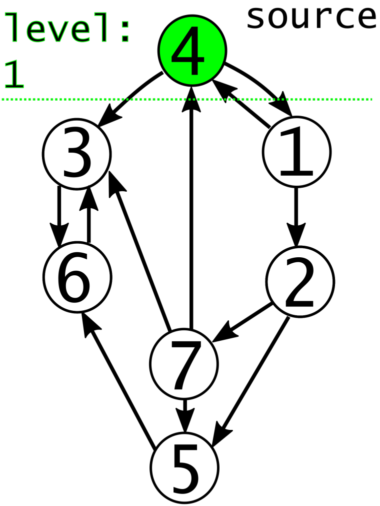



```
v = zeros (1,n) ;
q = false (1,n) ;
q (source) = true ;

v:     q:

0      .
0      .
0      .
0      4
0      .
0      .
0      .
```

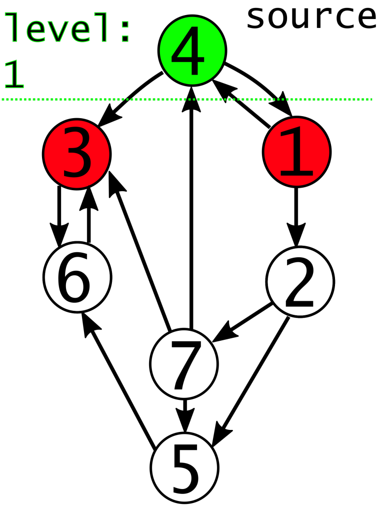
Breadth-first search: step 1a




```
v (q) = level ;
```

v:	q:
0	.
0	.
0	.
1	4
0	.
0	.
0	.

Breadth-first search: step 1b

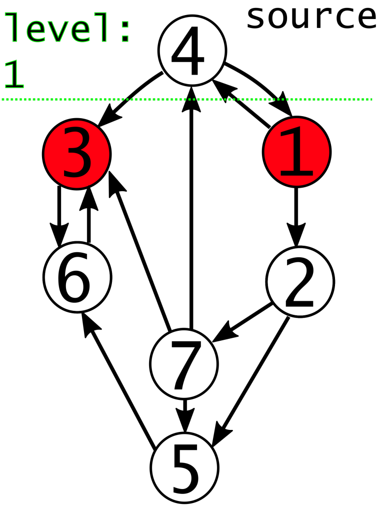



```
t = A*q ;
```

```
A                    *  q  =  t:

. . . 1 . . .           .     1
2 . . . . . .           .     .
. . . 3 . 3 3           .     3
4 . . . . . 4        *  4  =  .
. 5 . . . . 5           .     .
. . 6 . 6 . .           .     .
. 7 . . . . .           .     .
```

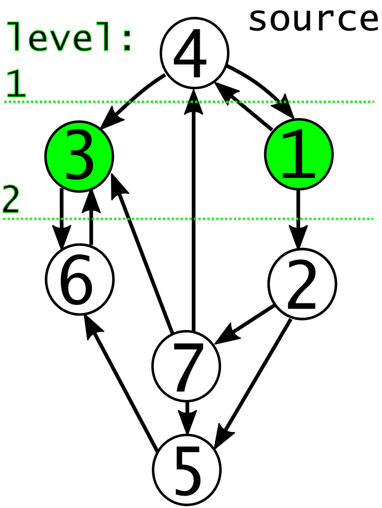
Breadth-first search: step 1c




```
q = false (1,n) ;
q (~v) = t ;
```

v:	t=A*q:	q(~v)=t
0	1	1
0	.	.
0	3	3
1	.	.
0	.	.
0	.	.
0	.	.

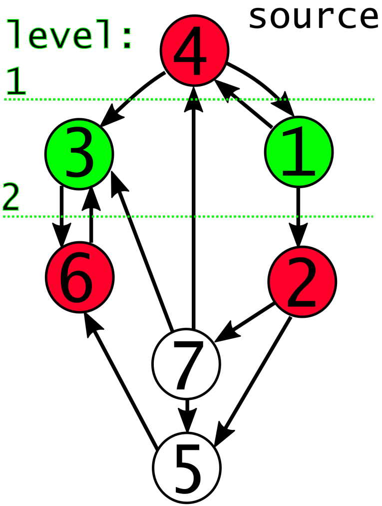
Breadth-first search: step 2a




```
v (q) = level ;
```

v:	q:
2	1
0	.
2	3
1	.
0	.
0	.
0	.

Breadth-first search: step 2b

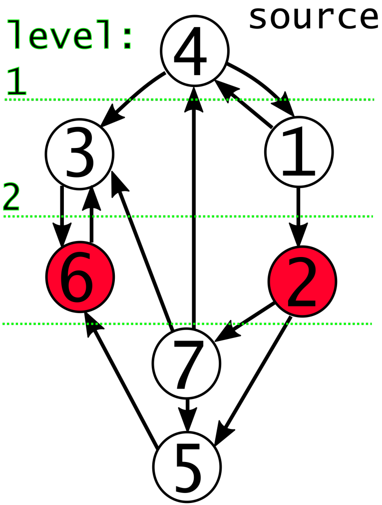



```
t = A*q ;
```

```
A                        *  q   =  t:

.  .  .  1  .  .  .         1        .
2  .  .  .  .  .  .         .        2
.  .  .  3  .  3  3         3        .
4  .  .  .  .  .  4   *     .   =    4
.  5  .  .  .  .  5         .        .
.  .  6  .  6  .  .         .        6
.  7  .  .  .  .  .         .        .
```

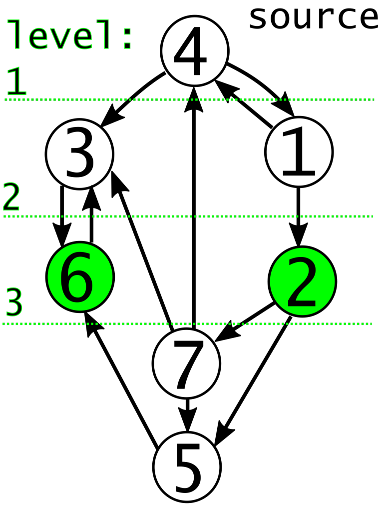
Breadth-first search: step 2c




```
q = false (1,n) ;
q (~v) = t ;
```

v:	t=A*q:	q(~v)=t
2	.	.
0	2	2
2	.	.
1	4	.
0	.	.
0	6	6
0	.	.

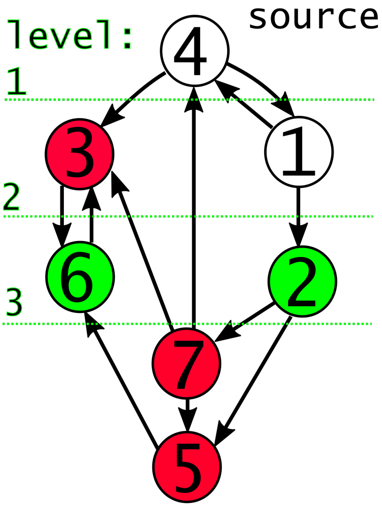
Breadth-first search: step 3a




```
v (q) = level ;
```

v:	q:
2	.
3	2
2	.
1	.
0	.
3	6
0	.

Breadth-first search: step 3b

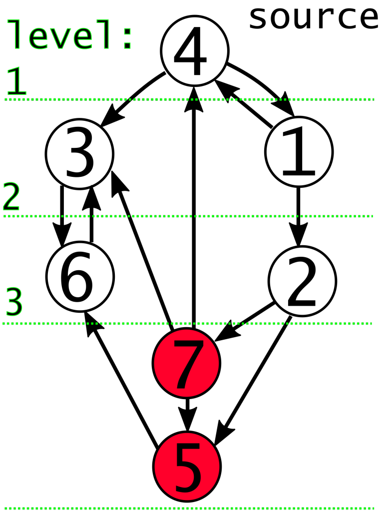



```
t = A*q ;
```

```
A                  *  q   =  t:

. . . 1 . . .         .      .
2 . . . . . .         2      .
. . . 3 . 3 3         .      3
4 . . . . . 4      *  .   =  .
. 5 . . . . 5         .      5
. . 6 . 6 . .         6      .
. 7 . . . . .         .      7
```

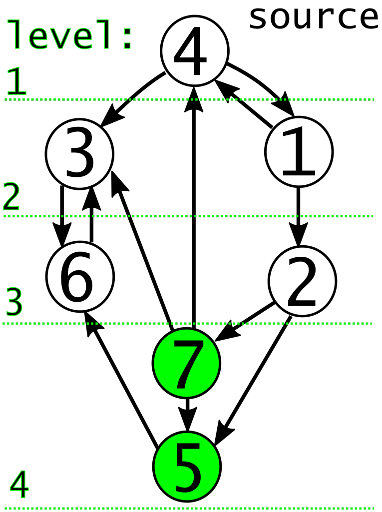
Breadth-first search: step 3c




```
q = false (1,n) ;
q (~v) = t ;
```

v:	t=A*q:	q(~v)=t
2	.	.
3	.	.
2	3	.
1	.	.
0	5	5
3	.	.
0	7	7

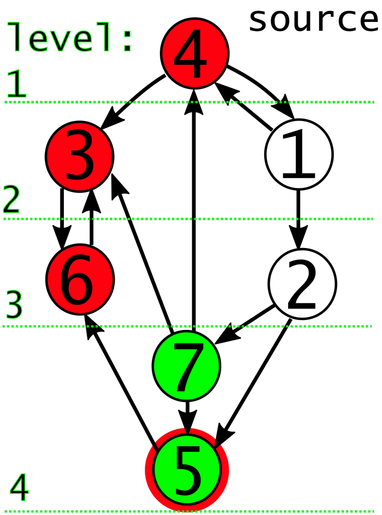
Breadth-first search: step 4a




```
v (q) = level ;
```

v:	q:
2	.
3	.
2	.
1	.
4	5
3	.
4	7

Breadth-first search: step 4b

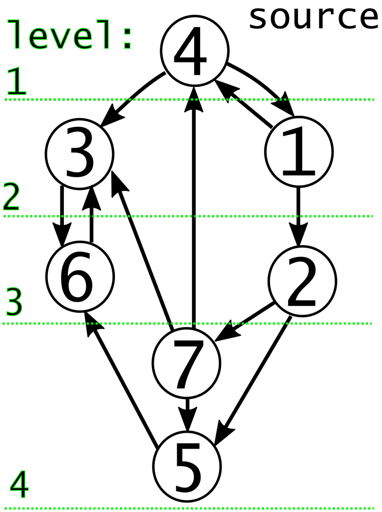



```
t = A*q ;


A                   *  q   =  t:

. . . 1 . . .          .      .
2 . . . . . .          .      .
. . . 3 . 3 3          .      3
4 . . . . . 4   *      .   =  4
. 5 . . . . 5          5      5
. . 6 . 6 . .          .      6
. 7 . . . . .          7      .
```

Breadth-first search: step 4c




```
q = false (1,n) ;
q (~v) = t ;
```

v:	t=A*q:	q(~v)=t
2	.	.
3	.	.
2	3	.
1	4	.
4	5	.
3	6	.
4	.	.

Luby's method for maximal independent set

```
iset = false (1,n) ;      % iset (i) = 1 if node i in output set
c = true (1,n) ;          % c (i) = 1 if node i is a candidate
while ( ... )
    % give each candidate a random score
    prob = zeros (1,n) ;
    prob(c) = some random score ;

    % new member if candidate score > max of its nieghbors
    neighbormax(c) = A * prob ;                % max-second semiring
    newmembers = prob(c) > neighbormax(c) ;

    % add new members to the independent set
    iset = iset | newmembers ;

    % remove new members from the candidate set
    c (~newmembers)= c & !newmembers ;

    % also remove neighbors of new members from candidate set
    newneighbors = false (1,n) ;
    newneighbors (c) = A * new_members ;    % or-and semiring
    c (~newneighbors) = c ;
```

GraphBLAS operations: overview

operation	MATLAB analog	GraphBLAS extras
matrix multiplication	`C=A*B`	960 built-in semirings
element-wise, set union	`C=A+B`	any operator
element-wise, set intersection	`C=A.*B`	any operator
reduction to vector or scalar	`s=sum(A)`	any operator
apply unary operator	`C=-A`	`C=f(A)`
transpose	`C=A'`	
submatrix extraction	`C=A(I,J)`	
submatrix assignment	`C(I,J)=A`	zombies and pending tuples

`C=A*B` with 960 built-in semirings, and each matrix one of 11 types: GraphBLAS has $960 \times 11^3 = 1,277,760$ built-in versions of matrix multiply. MATLAB has 4.

GraphBLAS objects

`GrB_Type`	11 built-in types, "any" user-defined type
`GrB_UnaryOp`	unary operator such as $z = -x$
`GrB_BinaryOp`	binary operator such as $z = x + y$
`GrB_Monoid`	associative operator like $z = x + y$ with identity 0
`GrB_Semiring`	a multiply operator and additive monoid
`GrB_Vector`	like an n-by-1 matrix
`GrB_Matrix`	a sparse m-by-n matrix
`GrB_Descriptor`	parameter settings

- All objects opaque
- matrix implemented as compressed-sparse column form, with sorted indices
- non-blocking mode; matrix can have pending operations

Accumulator and the Mask

- accumulator operator $Z = C \odot T$, like sparse matrix add (set union)

  for all entries (i,j) in $\mathbf{C} \cap \mathbf{T}$ (that is, entries in both $\mathbf{C}$ and $\mathbf{T}$)
  $\quad z_{ij} = c_{ij} \odot t_{ij}$
  for all entries (i,j) in $\mathbf{C} \setminus \mathbf{T}$ (that is, entries in $\mathbf{C}$ but not $\mathbf{T}$)
  $\quad z_{ij} = c_{ij}$
  for all entries (i,j) in $\mathbf{T} \setminus \mathbf{C}$ (that is, entries in $\mathbf{T}$ but not $\mathbf{C}$)
  $\quad z_{ij} = t_{ij}$

- Boolean mask matrix M controls what values are modified, just like MATLAB logical indexing. $M(i,j) = 1$ means $C(i,j)$ can be modified; $M(i,j) = 0$ leaves $C(i,j)$ untouched.

Accumulator and the Mask

- $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{T}$:

  if `accum` is `NULL`, $\mathbf{Z} = \mathbf{T}$; otherwise $\mathbf{Z} = \mathbf{C} \odot \mathbf{T}$
  if requested via descriptor (replace option), all entries cleared from $\mathbf{C}$
  if `Mask` is `NULL`
      $\mathbf{C} = \mathbf{Z}$ if `Mask` is not complemented; otherwise $\mathbf{C}$ is not modified
  else
      $\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{Z}$ if `Mask` is not complemented; otherwise $\mathbf{C}\langle\neg\mathbf{M}\rangle = \mathbf{Z}$

GraphBLAS notation

GrB_mxm	matrix-matrix multiply	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{AB}$
GrB_vxm	vector-matrix multiply	$\mathbf{w}'\langle\mathbf{m}'\rangle = \mathbf{w}' \odot \mathbf{u}'\mathbf{A}$
GrB_mxv	matrix-vector multiply	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{Au}$
GrB_eWiseMult	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \otimes \mathbf{B})$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \otimes \mathbf{v})$
GrB_eWiseAdd	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot (\mathbf{A} \oplus \mathbf{B})$
	set intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot (\mathbf{u} \oplus \mathbf{v})$
GrB_extract	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}(\mathbf{i},\mathbf{j})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot \mathbf{u}(\mathbf{i})$
GrB_assign	assign submatrix	$\mathbf{C}(\mathbf{i},\mathbf{j})\langle\mathbf{M}\rangle = \mathbf{C}(\mathbf{i},\mathbf{j}) \odot \mathbf{A}$
		$\mathbf{w}(\mathbf{i})\langle\mathbf{m}\rangle = \mathbf{w}(\mathbf{i}) \odot \mathbf{u}$
GrB_apply	apply unary operator	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot f(\mathbf{u})$
GrB_reduce	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{w} \odot [\oplus_j \mathbf{A}(:,j)]$
	reduce to scalar	$s = s \odot [\oplus_{ij} \mathbf{A}(i,j)]$
GrB_transpose	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{C} \odot \mathbf{A}'$

GraphBLAS performance: pending operations

- creating a matrix from list of tuples, same as MATLAB:

```
I = zeros (nz,1) ;
J = zeros (nz,1) ;
X = zeros (nz,1) ;
for k = 1:nz
    compute a value x, row index i, and column index j
    I (k) = i ;
    J (k) = j ;
    X (k) = x ;
end
A = sparse (I,J,X,m,n) ;
```

- just as fast in GraphBLAS (operations left pending), but painful in MATLAB:

```
A = sparse (m,n) ;   % an empty sparse matrix
for k = 1:nz
    compute a value x, row index i, and column index j
    A (i,j) = x ;
end
```

GraphBLAS performance: C(I,J)=A

- Submatrix assignment
- Example: `C` is the `Freescale2` matrix, 3 million by 3 million with 14.3 million nonzeros
- `I = randperm (n,5500)`
- `J = randperm (n,7000)`
- `A` = random sparse matrix with 38,500 nonzeros
- `C(I,J) = A`
    - 87 seconds in MATLAB
    - 0.74 seconds in GraphBLAS, *without* exploiting blocking mode

Zombies make C(I,J)=A fast

- *Zombie*: an entry marked for deletion but still in the data structure
  - suppose `C(i,j)` is present ("nonzero")
  - `C(i,j) = sparse(0)`
    - costly in MATLAB
    - GraphBLAS: turns turns `C(i,j)` into a *zombie*
  - remainder of matrix unchanged
  - `C(i,j) = sparse(x)` brings the zombie back to life
  - if `C` is used in another operation:
    killing a million zombies just as fast as killing one

Pending tuples make C(I,J)=A fast

- *Pending tuple*: an entry waiting to be added to the matrix
  - `C(i,j) = sparse(x)`
    - costly in MATLAB
    - GraphBLAS: goes into a list of pending tuples to be added later
  - remainder of matrix unchanged
  - if `C` is used in another operation:
    assembling a million tuples just as fast as adding one

Example: create a random matrix

```
GrB_Matrix_new (&A, GrB_FP64, nrows, ncols) ;
for (int64_t k = 0 ; k < ntuples ; k++)
{
    GrB_Index i = simple_rand_i ( ) % nrows ;
    GrB_Index j = simple_rand_i ( ) % ncols ;
    if (no_self_edges && (i == j)) continue ;
    double x = simple_rand_x ( ) ;
    // A (i,j) = x
    GrB_Matrix_setElement (A, i, j, x) ;
    if (make_symmetric)
    {
        // A (j,i) = x
        GrB_Matrix_setElement (A, j, i, x) ;
    }
}
```

Example: create a finite-element matrix

```
A = sparse (m,n) ; % create an empty n-by-n
                   % sparse GraphBLAS matrix

for i = 1:k
    construct a 8-by-8 sparse or dense finite-element F
    I and J define where the matrix F is to be added:
    I = a list of 8 row indices
    J = a list of 8 column indices
    % using GrB_assign, with the 'plus' accum operator:
    A (I,J) = A (I,J) + F
end
```

Example: equivalent of MATLAB wathen.m

```
GrB_Matrix_new (&F, GrB_FP64, 8, 8) ;
for (int j = 1 ; j <= ny ; j++) {
    for (int i = 1 ; i <= nx ; i++) {
        nn [0] = 3*j*nx + 2*i + 2*j + 1 ;
        nn [1] = nn [0] - 1 ;
        nn [2] = nn [1] - 1 ;
        nn [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        nn [5] = nn [4] + 1 ;
        nn [6] = nn [5] + 1 ;
        nn [7] = nn [3] + 1 ;
        for (int krow = 0 ; krow < 8 ; krow++) nn [krow]-- ;
        for (int krow = 0 ; krow < 8 ; krow++) {
            for (int kcol = 0 ; kcol < 8 ; kcol++) {
                // F (krow,kcol) = em (krow, kcol)
                GrB_Matrix_setElement (F, krow, kcol, em (krow,kcol
            }
        }
        // A (nn,nn) += F
        GrB_assign (A, NULL, GrB_PLUS_FP64, F, nn, 8, nn, 8, NULL)
    }
}
```

User-defined types, operators, monoids, semirings

- double complex: not native to GraphBLAS
- GraphBLAS is ≈ 26,500 lines of code
- adding complete suite of complex operators: 523 lines in "user" code
- any `typedef` with constant size can be added as a type
- example: `WildType`

```
typedef struct
{
    float stuff [4][4] ;
    char whatstuff [64] ;
}
wildtype ;                          // C version of wildtype
GrB_Type WildType ;                 // GraphBLAS version of wildtype
GrB_Type_new (&WildType, wildtype) ;
```

User-defined operator: add

```
void wildtype_add (wildtype *z, const wildtype *x, const wildtype *y)
{
    for (int i = 0 ; i < 4 ; i++)
    {
        for (int j = 0 ; j < 4 ; j++)
        {
            z->stuff [i][j] = x->stuff [i][j] + y->stuff [i][j] ;
        }
    }
    sprintf (z->whatstuff, "this was added") ;
    printf ("[%s] = [%s] + [%s]\n", z->whatstuff, x->whatstuff, y->whatstuff) ;
}

...

    // create the WildAdd operator
    GrB_BinaryOp WildAdd ;
    GrB_BinaryOp_new (&WildAdd, wildtype_add, WildType, WildType, WildType) ;
```

User-defined operator: multiply

```
void wildtype_mult (wildtype *z, const wildtype *x, const wildtype *y)
{
    for (int i = 0 ; i < 4 ; i++)
    {
        for (int j = 0 ; j < 4 ; j++)
        {
            z->stuff [i][j] = 0 ;
            for (int k = 0 ; k < 4 ; k++)
            {
                z->stuff [i][j] += (x->stuff [i][k] * y->stuff [k][j]) ;
            }
        }
    }
    sprintf (z->whatstuff, "this was multiplied") ;
    printf ("[%s] = [%s] * [%s]\n", z->whatstuff, x->whatstuff, y->whatstuff) ;
}

...

    // create the WildMult operator
    GrB_BinaryOp WildMult ;
    GrB_BinaryOp_new (&WildMult, wildtype_mult, WildType, WildType, WildType) ;
```

User-defined monoid and semiring

```
// create the WildAdder monoid
GrB_Monoid WildAdder ;
wildtype scalar_identity ;
for (int i = 0 ; i < 4 ; i++)
{
    for (int j = 0 ; j < 4 ; j++)
    {
        scalar_identity.stuff [i][j] = 0 ;
    }
}
sprintf (scalar_identity.whatstuff, "identity") ;
GrB_Monoid_UDT_new (&WildAdder, WildAdd, &scalar_identity) ;

// create the InTheWild semiring
GrB_Semiring InTheWild ;
GrB_Semiring_new (&InTheWild, WildAdder, WildMult) ;

// C = A*B
GrB_mxm (C, NULL, NULL, InTheWild, A, B, NULL) ;
```

Summary

- GraphBLAS: graph algorithms in the language of linear algebra
- "Sparse-anything" matrices, including user-defined types
- matrix multiplication with any semiring
- operations: `C=A*B`, `C=A+B`, reduction, transpose, accumulator/mask, submatrix extraction and assigment
- performance: most operations just as fast as MATLAB, submatrix assignment 100x or faster.
- Beta version available at suitesparse.com