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

Tim Davis
Texas A&M University

Oct 2, 2017

CSCE Seminar, Texas A&M University
Graph algorithms in the language of linear algebra

- Consider $C = A \ast B$ 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 $C = A \text{ or }. \text{and} B$ or $C = A \mid \& B$

- The GraphBLAS Spec: graphblas.org
- SuiteSparse:GraphBLAS implementation and performance
Breadth-first search in pseudo-MATLAB notation

\[ v = \text{zeros}(1,n) ; \quad \% \text{v}(k) \text{ is the BFS level (1 for source node)} \]
\[ q = \text{false}(1,n) ; \quad \% \text{boolean vector of size n} \]
\[ q(\text{source}) = \text{true} ; \quad \% q: \text{boolean vector of current level} \]

\begin{verbatim}
for level = 1:n
    v(q) = level ; \quad \% set v(i)=level where q(i) is true

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

    if (~any(q)) break ;
end
\end{verbatim}
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

\[
v = \text{zeros} \ (1,n) ; \\
q = \text{false} \ (1,n) ; \\
q \ (\text{source}) = \text{true} ; \\
\]

\[
v: \quad q: \\
0 \ . \\
0 \ . \\
0 \ . \\
0 \ 4 \\
0 \ . \\
0 \ . \\
0 \ . \\
0 \ .
\]
Breadth-first search: step 1a

\[ v(q) = \text{level} ; \]

\[
\begin{array}{c|c}
 v & q \\
\hline
 0 & . \\
 0 & . \\
 0 & . \\
 1 & 4 \\
 0 & . \\
 0 & . \\
 0 & . \\
\end{array}
\]
Breadth-first search: step 1b

t = A*q ;

\[ A \times q = t : \]

dots 1 dots 1 dots 1
2 dots dots dots dots
dots dots 3 dots 3 dots dots
4 dots dots dots 4 dots 4 = dots
5 dots dots dots 5 dots
dots dots dots 6 dots dots
dots dots dots dots dots
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

\[
v (q) = \text{level} ;
\]

\[
v : \\
2 1 \\
0 . \\
2 3 \\
1 . \\
0 . \\
0 .
\]
Breadth-first search: step 2b

\[ t = A*q ; \]

\[
\begin{array}{ccccccccc}
\ldots & 1 & \ldots & 1 & . & . & . & . & . \\
2 & \ldots & \ldots & . & 2 \\
\ldots & 3 & 3 & 3 & 3 & 3 & . & . & . \\
4 & \ldots & \ldots & 4 & * & . & = & 4 \\
. & 5 & \ldots & \ldots & 5 & . & . & . & . \\
. & . & 6 & 6 & . & . & . & 6 \\
. & 7 & \ldots & \ldots & \ldots & . & . & . & . \\
\end{array}
\]
Breadth-first search: step 2c

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

v: t=A*q: q(~v)=t

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
Breadth-first search: step 3a

\[ v(q) = \text{level} \]

\[
\begin{array}{c|c}
\text{v} & \text{q} \\
\hline
2 & . \\
3 & 2 \\
2 & . \\
1 & . \\
0 & . \\
3 & 6 \\
0 & . \\
\end{array}
\]
Breadth-first search: step 3b

t = A*q ;

\[
\begin{align*}
A & \quad * \quad q = \quad t : \\
\cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
2 & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
\cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
4 & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
\cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
5 & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
\cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
6 & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
\cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
7 & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \cdot \cdot & \cdot \\
\end{align*}
\]
Breadth-first search: step 3c

\[ q = \text{false } (1, n) ; \]
\[ q (\neg v) = t ; \]

\[ v: \quad t = A \cdot q: \quad q (\neg v) = t \]

<table>
<thead>
<tr>
<th>level</th>
<th>node</th>
<th>edge</th>
<th>node</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Edge weights:

- 1-3: 2
- 3-6: 3
- 6-2: 2
- 2-1: 3
- 1-0: 5
- 0-5: 5
- 5-0: 7
- 0-7: 7
Breadth-first search: step 4a

\[ v(q) = \text{level} ; \]

\[ v: \quad q:\]

2 .
3 .
2 .
1 .
4 5
3 .
4 7
Breadth-first search: step 4b

\[ t = A \cdot q \]

\[
\begin{array}{cccccccc}
\ldots & 1 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
2 & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\ldots & 3 & \ldots & 3 & 3 & \ldots & \ldots & 3 \\
4 & \ldots & \ldots & \ldots & 4 & \ldots & \ldots & \ldots \\
. & 5 & \ldots & \ldots & 5 & 5 & 5 & 5 \\
. & \ldots & 6 & 6 & \ldots & \ldots & 6 & \ldots \\
. & \ldots & \ldots & \ldots & \ldots & \ldots & 7 & \ldots \\
\end{array}
\]
Breadth-first search: step 4c

\[ q = false \ (1,n) ; \]
\[ q \ (\sim v) = t ; \]

\[
\begin{array}{cccc}
 v: & t = A * q: & q(\sim v) = t \\
 2 & . & . \\
 3 & . & . \\
 2 & 3 & . \\
 1 & 4 & . \\
 4 & 5 & . \\
 3 & 6 & . \\
 4 & . & . \\
\end{array}
\]
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 neighbors
    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

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

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

<table>
<thead>
<tr>
<th>GrB_Type</th>
<th>11 built-in types, “any” user-defined type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_UnaryOp</td>
<td>unary operator such as $z = -x$</td>
</tr>
<tr>
<td>GrB_BinaryOp</td>
<td>binary operator such as $z = x + y$</td>
</tr>
<tr>
<td>GrB_Monoid</td>
<td>associative operator like $z = x + y$ with identity 0</td>
</tr>
<tr>
<td>GrB_Semiring</td>
<td>a multiply operator and additive monoid</td>
</tr>
<tr>
<td>GrB_Vector</td>
<td>like an $n$-by-1 matrix</td>
</tr>
<tr>
<td>GrB_Matrix</td>
<td>a sparse $m$-by-$n$ matrix</td>
</tr>
<tr>
<td>GrB_Descriptor</td>
<td>parameter settings</td>
</tr>
</tbody>
</table>

- 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 $C \cap T$ (that is, entries in both $C$ and $T$)
  $$z_{ij} = c_{ij} \odot t_{ij}$$

  for all entries $(i, j)$ in $C \setminus T$ (that is, entries in $C$ but not $T$)
  $$z_{ij} = c_{ij}$$

  for all entries $(i, j)$ in $T \setminus C$ (that is, entries in $T$ but not $C$)
  $$z_{ij} = t_{ij}$$

- 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.
\( C\langle M \rangle = C \odot T: \)

if \text{accum} is NULL,  \( Z = T \); otherwise  \( Z = C \odot T \)

if requested via descriptor (replace option), all entries cleared from \( C \)

if \text{Mask} is NULL

\( C = Z \) if \text{Mask} is not complemented; otherwise \( C \) is not modified

else

\( C\langle M \rangle = Z \) if \text{Mask} is not complemented; otherwise  \( C\langle \neg M \rangle = Z \)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrB_mxm</td>
<td>matrix-matrix multiply</td>
<td>$C\langle M\rangle = C \odot AB$</td>
</tr>
<tr>
<td>GrB_vxm</td>
<td>vector-matrix multiply</td>
<td>$w'\langle m'\rangle = w' \odot u'A$</td>
</tr>
<tr>
<td>GrB_mxv</td>
<td>matrix-vector multiply</td>
<td>$w\langle m\rangle = w \odot Au$</td>
</tr>
<tr>
<td>GrB_eWiseMult</td>
<td>element-wise, set union</td>
<td>$C\langle M\rangle = C \odot (A \otimes B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$w\langle m\rangle = w \odot (u \otimes v)$</td>
</tr>
<tr>
<td>GrB_eWiseAdd</td>
<td>element-wise, set intersection</td>
<td>$C\langle M\rangle = C \odot (A \oplus B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$w\langle m\rangle = w \odot (u \oplus v)$</td>
</tr>
<tr>
<td>GrB_extract</td>
<td>extract submatrix</td>
<td>$C\langle M\rangle = C \odot A(i, j)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$w\langle m\rangle = w \odot u(i)$</td>
</tr>
<tr>
<td>GrB_assign</td>
<td>assign submatrix</td>
<td>$C(i, j)\langle M\rangle = C(i, j) \odot A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$w(i)\langle m\rangle = w(i) \odot u$</td>
</tr>
<tr>
<td>GrB_apply</td>
<td>apply unary operator</td>
<td>$C\langle M\rangle = C \odot f(A)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$w\langle m\rangle = w \odot f(u)$</td>
</tr>
<tr>
<td>GrB_reduce</td>
<td>reduce to vector</td>
<td>$w\langle m\rangle = w \odot [\oplus_j A(:, j)]$</td>
</tr>
<tr>
<td></td>
<td>reduce to scalar</td>
<td>$s = s \odot [\oplus_{ij} A(i, j)]$</td>
</tr>
<tr>
<td>GrB_transpose</td>
<td>transpose</td>
<td>$C\langle M\rangle = C \odot A'$</td>
</tr>
</tbody>
</table>
double complex: not native to GraphBLAS

GraphBLAS is $\approx 26,500$ 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) ;
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) ;
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);
Matrix data structure

- Primary component: CSC, compressed sparse column form
  - int64 p[ncols+1], column “pointers”
  - int64 i[nzmax], row indices, always sorted
  - void x[nzmax * sizeof(type)], “numerical” values
  - A(:,j): row indices i [p[j] ... p[j+1]-1], values in same place in x.
  - identical to MATLAB, except x is any type

- Any of p, i, x can be marked as shallow: pointer to content of another matrix

- pending operations: for C(I,J)=A
  - pending tuples
  - zombies
  - next and prev: pointers to place matrix in global queue of matrices with pending operations
Operations: \( C = A \times B \)

- phase 1: symbolic analysis
  - time is \( O(\text{flop count}) \), dynamic integer memory allocation
  - qsort of each column, unless result transposed
- phase 2: numeric
  - 960 built-in semirings:
    - 680 with multiplier \( T \times T \rightarrow T \)
      - 4 monoids: min, max, plus, times
      - 17 multiply operators: 1st, 2nd, min, max, plus, minus, times, div, iseq, isgt, islt, isge, isle, or, and, xor
      - \( T \): 10 non-boolean types (int and uint 8, 16, 32, 64; fp32, 64)
    - 240 with multiplier \( T \times T \rightarrow \text{bool} \)
      - 4 boolean monoids: and, or, xor, eq
      - 6 multiply operators: eq, ne, gt, lt, ge, le
      - \( T \): 10 non-boolean types (int and uint 8, 16, 32, 64; fp32, 64)
  - 40 purely boolean semirings
    - 4 boolean monoids: and, or, xor, eq
    - 10 multiply operators: 1st, 2nd, or, and, xor, eq, gt, lt, ge, le
Operations: \( C = A(I,J) \), submatrix extraction

- \( C = A(I,J) \), with row indices \( I \) and column indices \( J \)
- phase 1: symbolic analysis of \( I \)
  - invert \( I \) if large; \( i \)th bucket contains all \( k \) for which \( I(k) = i \)
  - find \( \max(I) \) and \( \min(I) \)
  - determine if result will need sorting
  - check special case: \( I = \text{imin} : \text{imax} \)
- phase 2: estimate size of \( C \), grows as needed
- phase 3: construct \( C \) one column \( j \) in \( J \) at a time
  - start with binary search for \( \text{imin} \)
  - 7 cases:
    1. \( \text{length}(I) = 1 \): binary search for \( A(i,j) \)
    2. \( I = \text{all rows} \): pure copy of \( A(:,j) \)
    3. \( I = \text{imin} : \text{imax} \): copy \( A(\text{imin} : \text{imax}) \)
    4. \( I \) is short vs \( \text{nnz}(A(:,j)) \): binary search for each \( i \) in \( I \)
    5. \( I \) is long compared with \( \text{nnz}(A(:,j)) \): scan all \( A(:,j) \) and lookup \( I \) inverse, then \( \text{qsort} \) the result
    6. as case 5, but \( \text{qsort} \) not needed; no duplicates in \( I \)
    7. as case 6, but \( \text{qsort} \) not needed; duplicates in \( I \)
Operations: \( C(I,J) = A \), submatrix assignment

- hardest function to implement
- modifies \( C \) in place
- costly to modify CSC format, so *zombies* and *pending tuples* are used
- zombies: entries still in CSC but marked for deletion
- pending tuples: unsorted list of entries to be added to CSC
- zombies and pending tuples can be left in the matrix for subsequent \( C(I,J) = A \)
S = C(I,J), symbolic analysis, S has pointers into C

For each column j in J
- sorted merge of S(:,j) and A(:,j)
- if entry is S but not A: entry in C becomes a zombie
- if entry not in S but in A: new entry is a pending tuple
- if in both: modify the entry in C

if non-blocking: done

if blocking
- delete all zombies
- C += build (pending tuples)
creating a matrix from list of tuples, same as MATLAB:

\[
I = \text{zeros} (\text{nz},1) ; \\
J = \text{zeros} (\text{nz},1) ; \\
X = \text{zeros} (\text{nz},1) ; \\
\text{for } k = 1:\text{nz} \\
\quad \text{compute a value } x, \text{ row index } i, \text{ and column index } j \\
\quad I (k) = i ; \\
\quad J (k) = j ; \\
\quad X (k) = x ; \\
\text{end} \\
A = \text{sparse} (I,J,X,m,n) ;
\]

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

\[
A = \text{sparse} (m,n) ; \quad \% \text{an empty sparse matrix} \\
\text{for } k = 1:\text{nz} \\
\quad \text{compute a value } x, \text{ row index } i, \text{ and column index } j \\
\quad A (i,j) = x ; \\
\text{end}
\]
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”)
  - $C(i,j) = \text{sparse}(0)$
    - costly in MATLAB
    - GraphBLAS: turns turns $C(i,j)$ into a zombie
  - remainder of matrix unchanged
  - $C(i,j) = \text{sparse}(x)$ brings the zombie back to life
  - if $C$ is used in another operation:
    - killing a million zombies just as fast as killing one
  - if $C$ is used in a subsequent $C(I,J)=A$: leave the zombies in place
Pending tuples make \( C(I,J) = A \) fast

- **Pending tuple**: an entry waiting to be added to the matrix
  - \( C(i,j) = \text{sparse}(x) \)
    - 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
  - if \( C \) is used in a subsequent \( C(I,J) = A \): leave the pending tuples in place
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

\begin{verbatim}
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
\end{verbatim}
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 [3] = (3*j-1)*nx + 2*j + i - 1 ;
        nn [4] = 3*(j-1)*nx + 2*i + 2*j - 3 ;
        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) ;
    }
}
Deviations from the spec

- size of mask for assign (only major deviation)
- additional built-in operators: binary comparators of form $T \times T \rightarrow T$, unary abs and one.
- new query functions/macros
  - `#define`'s for version, implementation
  - `GrB_Type_size`, like `sizeof(type)` in C
  - query type of inputs to unary and binary operators
  - query operator and identity of monoid
  - query monoid and multiply operator of semiring
  - query type of matrix and vector
  - query descriptor setting
- change of parameters
  - type parameter not needed for `GrB_Monoid_new`
  - added `nvals` for tuple extraction
  - order of parameters for `*Op_new` matches function signature
- `GrB_DEFAULT` for descriptor
- 44 predefined monoids
- 960 predefined semirings
- defined divide-by-zero and type-casting `Inf` to integer
Let \( C \) by \( n \)-by-\( n \) and let \( A \) and \( C(i,j) \) be \( k \)-by-\( k \) with \( O(k) \) entries. Suppose \( n \gg k \).

the spec: \( C\langle M\rangle(i,j) = C(i,j) \odot A \)

- Mask \( M \) same size as \( C \): \( n \)-by-\( n \)
- the “BigMask” option
- Odd, since only \( M(i,j) \) has any effect on the operation
  - if user has the BigMask already, no extra cost
  - if user has the LittleMask, then constructing the BigMask is \( \Omega(n) \) time and memory (slow!), but rest of assignment can be as low as \( \Omega(k) \).

SuiteSparse: \( C(i,j)\langle M\rangle = C(i,j) \odot A \)

- Mask \( M \) same size as \( A \) and \( C(i,j) \): \( k \)-by-\( k \)
- the “LittleMask” option
- all of \( M(i,j) \) effects on the operation; nothing else needed
  - if user has the BigMask, then \( \text{LittleMask} = \text{BigMask}(I,J) \) takes as little as \( \Omega(k) \) time to construct; no extra asymptotic cost
  - if user has the LittleMask, no extra work
Simple addition to the spec: \( I=imin:imax \)

- the spec has \( \text{GrB\_ALL} \)
- acts like the colon in the MATLAB \( C=A(:,J) \) or \( C(:,J)=A \)
- need method for specifying a range, \( C=A(imin:imax) \)
- consider a vector \( V \) of size \( n = 100 \) billion, with \( k = 100 \) entries
- \( C=V((n/2):n) \) takes \( O(n) \) time since \( I=(n/2):n \) must be explicitly constructed and parsed (\textit{slow!}); remainder of work is \( O(k) \).
- \text{SuiteSparse:GraphBLAS} detects if \( I \) is a contiguous range, \( I=imin:imax \), and both assign and extract operations exploit this important special case
- propose \( I=[imin \text{ imax}] \) of size 2, with \( ni=\text{GrB\_RANGE} \)
- could be extended to \( I=imin:stride:imax \)
<table>
<thead>
<tr>
<th>Code size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>include files</td>
<td>6,284</td>
</tr>
<tr>
<td>C(I,J)=A</td>
<td>3,445</td>
</tr>
<tr>
<td>objects</td>
<td>3,585</td>
</tr>
<tr>
<td>C=A*B</td>
<td>2,619</td>
</tr>
<tr>
<td>support</td>
<td>1,904</td>
</tr>
<tr>
<td>A=C(I,J)</td>
<td>1,893</td>
</tr>
<tr>
<td>operators</td>
<td>1,518</td>
</tr>
<tr>
<td>ewise</td>
<td>1,414</td>
</tr>
<tr>
<td>build</td>
<td>1,039</td>
</tr>
<tr>
<td>reduce</td>
<td>857</td>
</tr>
<tr>
<td>accumulator/mask</td>
<td>794</td>
</tr>
<tr>
<td>transpose</td>
<td>666</td>
</tr>
<tr>
<td>apply</td>
<td>216</td>
</tr>
<tr>
<td>typecast</td>
<td>251</td>
</tr>
<tr>
<td>total library</td>
<td>26,503</td>
</tr>
<tr>
<td>testing</td>
<td>25,331</td>
</tr>
<tr>
<td>demo</td>
<td>3,498</td>
</tr>
<tr>
<td>overall total</td>
<td>55,332</td>
</tr>
<tr>
<td>package</td>
<td>library &amp; demo</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>GraphBLAS</td>
<td>26,503</td>
</tr>
<tr>
<td>UMFPACK</td>
<td>39,485</td>
</tr>
<tr>
<td>CHOLMOD</td>
<td>45,467</td>
</tr>
<tr>
<td>CSparse</td>
<td>2,247</td>
</tr>
<tr>
<td>SPQR</td>
<td>20,780</td>
</tr>
<tr>
<td>KLU</td>
<td>7,976</td>
</tr>
</tbody>
</table>

GraphBLAS testing: each function implemented twice: C for speed, and simple MATLAB m-file to test for correctness
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 assignment
performance: most operations just as fast as MATLAB, submatrix assignment 100x or faster.
Beta version 0.2.0 available at suitesparse.com
Essentially compliant with the spec
- added new functions, #defines, operators, predefined monoids and semirings
- minor changes to some parameters
- major change to mask for assign operation
These slides are combined from two very different talks:

- **SuiteSparse:GraphBLAS: graph algorithms via sparse matrix operations on semirings**, Sparse Days at CERFACS, Toulouse, France, Sept 6-8, 2017, http://cerfacs.fr/en/actualite/sparse-days-meeting-2017-at-cerfacs/. This talk gave a detailed description of GraphBLAS itself, with examples (including the breadth-first-search in these slides). Only brief details of SuiteSparse:GraphBLAS implementation and performance were given.

- **SuiteSparse:GraphBLAS: a complete implementation of the GraphBLAS specification**, 2017 IEEE High Performance Extreme Computing Conference (HPEC ’17), Waltham, Massachusetts, Sept 12-14, 2017. http://www.ieee-hpec.org. This invited talk was part of a GraphBLAS forum, so very few details of GraphBLAS were given. Instead, the talk focused on the SuiteSparse:GraphBLAS implementation of the GraphBLAS specification, and its performance.

These slides combine both topics into a single talk.