Graph algorithms, computational motifs, and GraphBLAS

Aydın Buluç (abuluc@lbl.gov)
LBNL and UC Berkeley

Joint work with Ariful Azad, Jeremy Kepner, Tim Mattson, Jose Moreira, Scott McMillan, Carl Yang

ECP Annual Meeting
February 7, 2018
Outline

• Motivation

• Case studies:
  
  A. **Graph traversals**: Breadth-first search
     • Motif: Sparse matrix times sparse vector (SpMSpV)
  
  B. **Maximal Independent Sets**: Luby’s algorithm
     • Motif: SpMSpV
  
  C. **Triangle Counting**
     • Motif: SpGEMM
  
  D. **Betweenness Centrality (optional)**
     • Motif: SpMSpV or sparse matrix-matrix multiply (SpGEMM)
Large Graphs in Scientific Computing

Matching in bipartite graphs: Permuting to heavy diagonal or block triangular form

Graph partitioning: *Dynamic load balancing* in parallel simulations

Picture (left) credit: Sanders and Schulz

Problem size: as big as the sparse linear system to be solved or the simulation to be performed
**Manifold Learning**

**Isomap (Nonlinear dimensionality reduction):** Preserves the intrinsic geometry of the data by using the geodesic distances on manifold between all pairs of points

**Tools used or desired:**
- K-nearest neighbors
- *All pairs shortest paths (APSP)*
- Top-k eigenvalues

---

Large Graphs in Biology

Whole genome assembly

A Read Layout
R₁: GACCTACA
R₂: ACCTACAA
R₃: CCTACAAG
R₄: CTACAAGT
A: TACAAGTT
B: ACAAGTGA
C: CAAGTTAG
X: TACAAGTC
Y: ACAAGTCC
Z: CAAGTCCG

B Overlap Graph

Vertices: reads

C de Bruijn Graph

Vertices: k-mers

26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploidy wheat genome W7984 for k=51

Potentially millions of neurons and billions of edges with developing technologies

Schatz et al. (2010) Perspective: Assembly of Large Genomes w/2nd-Gen Seq. Genome Res. (figure reference)
The case for sparse matrices

Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

<table>
<thead>
<tr>
<th>Traditional graph computations</th>
<th>Graphs in the language of linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data driven, unpredictable communication.</td>
<td>Fixed communication patterns</td>
</tr>
<tr>
<td>Irregular and unstructured, poor locality of reference</td>
<td>Operations on matrix blocks exploit memory hierarchy</td>
</tr>
<tr>
<td>Fine grained data accesses, dominated by latency</td>
<td>Coarse grained parallelism, bandwidth limited</td>
</tr>
</tbody>
</table>
Linear-algebraic primitives for graphs

Sparse matrix $X$ sparse matrix

Sparse matrix $X$ sparse vector

Element-wise operations

Sparse matrix indexing

Is **think-like-a-vertex** really more productive?

“Our mission is to build up a linear algebra sense to the extent that vector-level thinking becomes as natural as scalar-level thinking.”

- Charles Van Loan
Examples of semirings in graph algorithms

<table>
<thead>
<tr>
<th>Semiring</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real field: $(\mathbb{R}, +, \times)$</td>
<td>Classical numerical linear algebra</td>
</tr>
<tr>
<td>Boolean algebra: $({0, 1},</td>
<td>, &amp;)$</td>
</tr>
<tr>
<td>Tropical semiring: $(\mathbb{R} \cup {\infty}, \min, +)$</td>
<td>Shortest paths</td>
</tr>
<tr>
<td>$(S, \text{select, select})$</td>
<td>Select subgraph, or contract nodes to form quotient graph</td>
</tr>
<tr>
<td>$(\text{edge/vertex attributes, vertex data aggregation, edge data processing})$</td>
<td>Schema for user-specified computation at vertices and edges</td>
</tr>
<tr>
<td>$(\mathbb{R}, \max, +)$</td>
<td>Graph matching &amp; network alignment</td>
</tr>
<tr>
<td>$(\mathbb{R}, \min, \times)$</td>
<td>Maximal independent set</td>
</tr>
</tbody>
</table>

- **Shortened semiring notation:** $(\text{Set, Add, Multiply})$. Both identities omitted.
- **Add:** Traverses edges, **Multiply:** Combines edges/paths at a vertex
- Neither add nor multiply needs to have an inverse.
- Both **add** and **multiply** are **associative**, **multiply distributes** over **add**
Graph Algorithms on GraphBLAS

**GraphBLAS primitives in increasing arithmetic intensity**

- Sparse Matrix-Sparse Vector (SpMSpV)
- Sparse Matrix-Dense Vector (SpMV)
- Sparse Matrix Times Multiple Dense Vectors (SpMM)
- Sparse - Sparse Matrix Product (SpGEMM)
- Sparse - Dense Matrix Product (SpDM³)

**Miscellaneous:**
- Connectivity, traversal (BFS), independent sets (MIS), graph matching

**Centrality:**
- (PageRank, betweenness, closeness)

**Graph clustering:**
- (Markov cluster, peer pressure, spectral, local)

**Shortest paths:**
- (all-pairs, single-source, temporal)

**Miscellaneous:**
- Connectivity, traversal (BFS), independent sets (MIS), graph matching

- Centrality (PageRank, betweenness, closeness)
- Graph clustering (Markov cluster, peer pressure, spectral, local)
- Shortest paths (all-pairs, single-source, temporal)

http://graphblas.org
Markov Cluster Algorithm (MCL)

Widely popular and successful algorithm for discovering clusters in protein interaction and protein similarity networks

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
MCL process is both **computationally expensive** and **memory hungry**, limiting the sizes of networks that can be clustered

- HipMCL overcomes such limitation via **sparse parallel algorithms**.
- **Up to 1000X times faster** than original MCL with same accuracy.
- Easily clusters a network of ~75M nodes with ~68B edges in ~2.4 hours using ~2000 nodes of Cori/NERSC.

Machine Learning on [Graph]BLAS

Higher-level machine learning tasks

Logistic Regression, Support Vector Machines
Dimensionality Reduction (NMF, CX, PCA)
Clustering (e.g., MCL, Spectral Clustering)
Partial Correlation Estimation (CONCORD)
Deep Learning (Neural Nets)

Sparse Matrix-Sparse Vector (SpMSpV)
Sparse Matrix-Dense Vector (SpMV)
Sparse Matrix-Multiple Dense Vectors (SpMM)
Sparse x Sparse Matrix (SpGEMM)
Dense Matrix-Vector (BLAS2)
Sparse x Dense Matrix (SpDM³)
Dense Matrix-Matrix (BLAS3)

Graph/Sparse/Dense BLAS functions (in increasing arithmetic intensity)
Graph traversal: Breadth-first search (BFS)

Memory requirements (# of machine words):
- Sparse graph representation: $m+n$
- Stack of visited vertices: $n$
- Distance array: $n$

Breadth-first search is a very important building block for other parallel graph algorithms such as (bipartite) matching, maximum flow, (strongly) connected components, betweenness centrality, etc.
Breadth-first search using matrix algebra from to $A^T$
Replace scalar operations
*Multiply* -> select
*Add* -> minimum

from

parents:

1

2

3

4

5

6

7

1

7

A^T

X

A^T X
Select vertex with minimum label as parent

parents:

1 4
2
1
4
2
2
2
2

from

to

A^T

X

A^TX
parents:

\[
\begin{bmatrix}
1 \\
4 \\
1 \\
2 \\
3 \\
2 \\
7 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 \\
7 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
A^T \\
X \\
A^TX \\
\end{bmatrix}
\]

from

to
Breadth-First Search in GraphBLAS

GrB_Vector q;
GrB_Vector_new(&q, GrB_BOOL, n); // Vector<bool> q(n) = false
GrB_Vector_setElement(q, (bool)true, s); // q[s] = true, false everywhere else

GrB_Monoid Lor; // Logical—or monoid
GrB_Monoid_new(&Lor, GrB_LOR, false);

GrB_Semiring Boolean; // Boolean semiring
GrB_Semiring_new(&Boolean, Lor, GrB_LAND);

GrB_Descriptor desc; // Descriptor for vxm
GrB_Descriptor_new(&desc);
GrB_Descriptor_set(desc, GrB_MASK, GrB_SCMP); // invert the mask
GrB_Descriptor_set(desc, GrB_OUTP, GrB_REPLACE); // clear the output before assignment

GrB_UnaryOp apply_level;
GrB_UnaryOp_new(&apply_level, return_level, GrB_INT32, GrB_BOOL);

/*
 * BFS traversal and label the vertices.
 */
GrB_Index nvals;
do {
    ++level; // next level (start with 1)
    GrB_apply(*v, GrB_NULL, GrB_PLUS_INT32, apply_level, q, GrB_NULL); // v[q] = level
    GrB_vxm(q, *v, GrB_NULL, Boolean, q, A, desc); // q[!v] = q || .&& A ; finds all the
    // unvisited successors from current q
    GrB_Vector_nvals(&nvals, q); // if there is no successor in q, we are done.
} while (nvals);
Maximal Independent Set

- Graph with vertices $V = \{1,2,\ldots,n\}$
- A set $S$ of vertices is independent if no two vertices in $S$ are neighbors.
- An independent set $S$ is maximal if it is impossible to add another vertex and stay independent.
- An independent set $S$ is maximum if no other independent set has more vertices.
- Finding a maximum independent set is intractably difficult (NP-hard).
- Finding a maximal independent set is easy, at least on one processor.

The set of red vertices $S = \{4,5\}$ is independent and is maximal but not maximum.
Parallel, Randomized MIS Algorithm

1. S = empty set; C = V;
2. while C is not empty {
3. label each v in C with a random r(v);
4. for all v in C in parallel {
5. if r(v) < min( r(neighbors of v) ) {
6. move v from C to S;
7. remove neighbors of v from C;
8. }
9. }
10. }

S = { }
C = { 1, 2, 3, 4, 5, 6, 7, 8 }

1. \( S = \) empty set; \( C = V; \)
2. \( \text{while } C \text{ is not empty } \{ \)
3. \( \text{label each } v \text{ in } C \text{ with a random } r(v); \)
4. \( \text{for all } v \text{ in } C \text{ in parallel } \{ \)
5. \( \text{if } r(v) < \min( r(\text{neighbors of } v) ) \} \)
6. \( \text{move } v \text{ from } C \text{ to } S; \)
7. \( \text{remove neighbors of } v \text{ from } C; \)
8. \( \} \)
9. \( \} \)
10. \( } \)

\( S = \{ \} \)
\( C = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \)
1. \( S = \text{empty set}; \ C = V; \)
2. \( \text{while } C \text{ is not empty} \{ \)
3. \( \text{label each } v \text{ in } C \text{ with a random } r(v); \)
4. \( \text{for all } v \text{ in } C \text{ in parallel} \{ \)
5. \( \text{if } r(v) < \min( r(\text{neighbors of } v) ) \{ \)
6. \( \text{move } v \text{ from } C \text{ to } S; \)
7. \( \text{remove neighbors of } v \text{ from } C; \)
8. \( \}\)
9. \( \}\)
10. \( \}\)
1. \( S = \text{empty set}; \ C = V; \)
2. \( \text{while} \ C \ \text{is not empty} \ { \}
3. \( \text{label each} \ v \ \text{in} \ C \ \text{with a random} \ r(v); \)
4. \( \text{for all} \ v \ \text{in} \ C \ \text{in parallel} \ { \}
5. \( \text{if} \ r(v) < \min( r(\text{neighbors of} \ v) ) \ { \}
6. \( \text{move} \ v \ \text{from} \ C \ \text{to} \ S; \)
7. \( \text{remove neighbors of} \ v \ \text{from} \ C; \)
8. \( \}
9. \( \}
10. \}
1. $S$ = empty set; $C$ = $V$;
2. while $C$ is not empty {
3. label each $v$ in $C$ with a random $r(v)$;
4. for all $v$ in $C$ in parallel {
5. if $r(v)$ < min( $r$(neighbors of $v$) ) {
6. move $v$ from $C$ to $S$;
7. remove neighbors of $v$ from $C$;
8. }
9. }
10. }

$S = \{1, 5\}$
$C = \{6, 8\}$
Parallel, Randomized MIS Algorithm

1. \( S = \text{empty set}; \ C = V; \)
2. \[ \text{while } C \text{ is not empty} \{ \]
3. \[ \text{label each } v \text{ in } C \text{ with a random } r(v); \]
4. \[ \text{for all } v \text{ in } C \text{ in parallel} \{ \]
5. \[ \text{if } r(v) < \min( r(\text{neighbors of } v) ) \{ \]
6. \[ \text{move } v \text{ from } C \text{ to } S; \]
7. \[ \text{remove neighbors of } v \text{ from } C; \]
8. \] 9. \] 10. \}
1. $S = \text{empty set}; \quad C = V$;
2. while $C$ is not empty {
3. label each $v$ in $C$ with a random $r(v)$;
4. for all $v$ in $C$ in parallel {
5. if $r(v) < \min( r(\text{neighbors of } v) )$ {
6. move $v$ from $C$ to $S$;
7. remove neighbors of $v$ from $C$;
8. }
9. }
10. }

**Theorem:** This algorithm “very probably” finishes within $O(\log n)$ rounds.

*work $\sim O(n \log n)$, but span $\sim O(\log n)$*
A Variant of Luby’s Algorithm in GraphBLAS

// Iterate while there are candidates to check.
GrB_Index nvals;
GrB_Vector_nvals(&nvals, candidates);
while (nvals > 0) {
    // compute a random probability scaled by inverse of degree
    GrB_apply(prob, candidates, GrB_NULL, set_random, degrees, r_desc);

    // compute the max probability of all neighbors
    GrB_mxv(neighbor_max, candidates, GrB_NULL, maxSelect2nd, A, prob, r_desc);

    // select vertex if its probability is larger than all its active neighbors,
    // and apply a ”masked no-op” to remove stored falses
    GrB_eWiseAdd(new_members, GrB_NULL, GrB_NULL, GrB_GT_FP64, prob, neighbor_max, GrB_NULL);
    GrB_apply(new_members, new_members, GrB_NULL, GrB_IDENTITY_BOOL, new_members, r_desc);

    // add new members to independent set.
    GrB_eWiseAdd(*iset, GrB_NULL, GrB_NULL, GrB_LOR, *iset, new_members, GrB_NULL);

    // remove new members from set of candidates c = c & !new
    GrB_eWiseMult(candidates, new_members, GrB_NULL,
                   GrB_LAND, candidates, candidates, sr_desc);

    GrB_Vector_nvals(&nvals, candidates);
    if (nvals == 0) { break; } // early exit condition

    // Neighbors of new members can also be removed from candidates
    GrB_mxv(new_neighbors, candidates, GrB_NULL, Boolean, A, new_members, GrB_NULL);
    GrB_eWiseMult(candidates, new_neighbors, GrB_NULL,
                  GrB_LAND, candidates, candidates, sr_desc);
}

http://graphblas.org
Clustering coefficient:
- $\Pr (\text{wedge } i-j-k \text{ makes a triangle with edge } i-k)$
- $3 \times \frac{\# \text{ triangles}}{\# \text{ wedges}}$
- $3 \times \frac{4}{19} = 0.63$ in example
- may want to compute for each vertex $j$

Cohen’s algorithm to count triangles:
- Count triangles by lowest-degree vertex.
- Enumerate “low-hinged” wedges.
- Keep wedges that close.
Counting triangles

\[ A = L + U \]  
\[ (\text{hi-} \to \text{lo} + \text{lo-} \to \text{hi}) \]

\[ L \times U = B \]  
\[ (\text{wedge, low hinge}) \]

\[ A \wedge B = C \]  
\[ (\text{closed wedge}) \]

\[ \text{sum}(C)/2 = 4 \text{ triangles} \]
Triangle Counting in GraphBLAS

/* * Given, L, the lower triangular portion of n x n adjacency matrix A (of and * undirected graph), computes the number of triangles in the graph. */

uint64_t triangle_count(GrB_Matrix L) // L: NxN, lower-triangular, bool
{
    GrB_Index n;
    GrB_Matrix_nrows(&n, L); // n = # of vertices

    GrB_Matrix C;
    GrB_Matrix_new(&C, GrB_UINT64, n, n);

    GrB_Monoid UInt64Plus; // integer plus monoid
    GrB_Monoid_new(&UInt64Plus, GrB_PLUS_UINT64, 0 ul);

    GrB_Semiring UInt64Arithmetic; // integer arithmetic semiring
    GrB_Semiring_new(&UInt64Arithmetic, UInt64Plus, GrB.TIMES_UINT64);

    GrB_Descriptor desc_tb; // Descriptor for mmx
    GrB_Descriptor_new(&desc_tb);
    GrB_Descriptor_set(desc_tb, GrB_INP1, GrB_TRAN); // transpose the second matrix

    GrB_mmx(C, L, GrB_NULL, UInt64Arithmetic, L, L, desc_tb); // C</L> = L .+ L'

    uint64_t count;
    GrB_reduce(&count, GrB_NULL, UInt64Plus, C, GrB_NULL); // 1-norm of C

    GrB_free(&C); // C matrix no longer needed
    GrB_free(&UInt64Arithmetic); // Semiring no longer needed
    GrB_free(&UInt64Plus); // Monoid no longer needed
    GrB_free(&desc_tb); // descriptor no longer needed

    return count;
}
Conclusions

- GraphBLAS enables one to efficiently cast graph algorithms and machine learning methods into the languages of sparse matrices.
- While elegant and efficient for problems that fit into the linear algebra framework, it is admittedly not fully universal.
- The standard definition by the C API group and a compliant implementation by Tim Davis available at http://graphblas.org
- More parallel implementations in the works. Currently one can use approximate GraphBLAS implementations from Combinatorial BLAS, Kokkos, and Cyclops Tensor Framework.
Betweenness Centrality

Definition:

$C_B(v)$: Among all the shortest paths, what fraction of them pass through the node of interest?

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

$\sigma_{st}$ is the number of shortest paths between vertices $s$ and $t$

$\sigma_{st}(v)$ is the number of such paths that pass through vertex $v$

- APSP is wasteful for sparse graphs
- Brandes’ algorithm is $O(mn)$ for unweighted graphs
### Betweenness Centrality: Data Structures

- Pick a starting vertex (4)
- Initialize vectors: $q$, $\tilde{q}$, and $t_d$
Betweenness Centrality: Get Neighbors

\[ A^T \]

\[ \tilde{q} \rightarrow (A^T \tilde{q}).\neg q \]

- Get 2nd neighbors from starting vertex: \( A^T \tilde{q} \)
- Eliminate existing vertices: \( .\neg q \)
- Tally: \( q += \tilde{q} \)
- Update table: \( t_2 = \tilde{q} \)
Betweenness Centrality: Get Neighbors

- Get 3rd neighbors from starting vertex: $A^T \tilde{q}$; sum paths to vertex
- Eliminate existing vertices: $\cdot \star \neg q$
- Tally: $q += q$
- Update table: $t_2 = q$
Betweenness Centrality: Get Neighbors

- Get 4th neighbors from starting vertex: $A^T\tilde{q}$
- Eliminate existing vertices: $\cdot \ast \neg q$
- Tally: $q+ = \tilde{q}$
- Update table: $t_2 = \tilde{q}$
Driver: Multiple-source breadth-first search

- Sparse array representation => space efficient
- Sparse matrix-matrix multiplication => work efficient
- Three possible levels of parallelism: searches, vertices, edges
- Highly-parallel implementation for Betweenness Centrality*
  
  *: A measure of influence in graphs, based on shortest paths
#include "GraphBLAS.h"

GrB_Info BC_update(GrB_Vector *delta, GrB_Matrix A, GrB_Index *s, GrB_Index nsver) 
{
    GrB_Index n;
    GrB_Matrix_nrows(&n, A); // n = # of vertices in graph
    GrB_Vector_new(delta,GrB_FP32,n); // Vector<float> delta(n)
    GrB_Monoid Int32Add; // Monoid <int32_t,+,-,0>
    GrB_Monoid_new(&Int32Add,GrB_INT32,GrB_PLUS_INT32,0);
    GrB_Semiring Int32AddMul; // Semiring <int32_t,int32_t,int32_t,+,*,-,0>
    GrB_Semiring_new(&Int32AddMul,Int32Add,GrB_TIMES_INT32);

    GrB_Descriptor desc_tsr; // Descriptor for BFS phase mxm
    GrB_Descriptor_new(&desc_tsr);
    GrB_Descriptor_set(desc_tsr,GrB_INP0,GrB_TRAN); // transpose of the adjacency matrix
    GrB_Descriptor_set(desc_tsr,GrB_MASK,GrB_SCMP); // structural complement of the mask
    GrB_Descriptor_set(desc_tsr,GrB_OUTP,GrB_REPLACE); // clear output before result is stored

    // index and value arrays needed to build numsp
    GrB_Index *i_nsver = malloc(sizeof(GrB_Index)*nsver);
    int32_t *ones = malloc(sizeof(int32_t)*nsver);
    for(int i=0; i<nsver; ++i) {
        i_nsver[i] = i;
        ones[i] = 1;
    }
    ...
}
Forward sweep of BC in GraphBLAS C API

... GrB_Matrix numsp; // Its nonzero structure holds all vertices that have been discovered GrB_Matrix_new(&numsp, GrB_INT32, n, nsver); // also stores # of shortest paths so far

GrB_Matrix_build(&numsp,GrB_NULL,GrB_NULL,s,i_nsver,ones,nsver,GrB_PLUS_INT32,GrB_NULL);
free(i_nsver); free(ones);

GrB_Matrix frontier; // Holds the current frontier where values are path counts.
GrB_Matrix_new(&frontier, GrB_INT32, n, nsver); // Initialized: neighbors of each source
GrB_extract(&frontier,numsp,GrB_NULL,A,GrB_ALL,n,s,nsver,desc_tsr);

// The memory for an entry in sigmas is only allocated within the do-while loop if needed
GrB_Matrix *sigmas = malloc(sizeof(GrB_Matrix)*n); // n is an upper bound on diameter
int32_t d = 0; // BFS level number
int32_t nvals = 0; // nvals == 0 when BFS phase is complete
do {
    // --------------------- The BFS phase (forward sweep) --------------------------
    GrB_Matrix_new(&(sigmas[d]), GrB_BOOL, n, nsver);
    // sigmas[d][:,s] = d^th level frontier from source vertex s

    GrB_apply(&(sigmas[d]),GrB_NULL,GrB_NULL,GrB_IDENTITY_BOOL,frontier,GrB_NULL);
    GrB_eWiseAdd(&numsp,GrB_NULL,GrB_NULL,Int32Add,numsp,frontier,GrB_NULL);
    // numsp += frontier (accum path counts)

    GrB_mxm(&frontier,numsp,GrB_NULL,Int32AddMul,A,frontier,desc_tsr);
    // f<!numsp> = A' +.* f (update frontier)
    GrB_Matrix_nvals(&nvals,frontier)
    d++;
} while (nvals);

...
Forward sweep of BC in GraphBLAS C API

- The `GrB_mxm` call forms the next frontier in one step by both expanding the current frontier (i.e., discovering the 1-hop neighbors of the set of vertices in the current frontier) and pruning the vertices that have already been discovered.
- The former is achieved by setting the descriptor, `desc_tsr`, to use the transpose of the adjacency matrix. The latter is achieved by setting the descriptor to use the structural complement of the mask and by passing the `numsp` matrix as the mask parameter.
- The implicit cast of `numsp` to `Boolean` allows `GrB_mxm` to interpret `numsp` as the set of previously discovered vertices.
- Note that the descriptor is also set to `GrB_REPLACE` to ensure that the frontier is overwritten with new values.

```c
GrB_mxms(&frontier,numsp,GrB_NULL,Int32AddMul,A,frontier,desc_tsr);
// f<!numsp> = A' +.* f (update frontier)
GrB_Matrix_nvals(&nvals,frontier)
```

Betweenness Centrality: Roll back & Tally

- Initialize the centrality update: $\tilde{c}$
- Will hold the contributions of these shortest paths to each vertexes betweenness centrality
Betweenness Centrality: Roll back & Tally

- Select 4th neighbors, divide by number of paths to these nodes:
  
  \[
  (1+c) \cdot t_4 / q = w
  \]
Betweenness Centrality: Roll back & Tally

- Select 4th neighbors, divide by number of paths to these nodes:
  \[(1+c) \cdot t_4 / q = w\]
- Find 3rd neighbors: \(Aw\)
Betweenness Centrality: Roll back & Tally

\[ A \cdot (1+c) \cdot \frac{t_4}{q} = Aw \cdot (q \cdot t_3) \]

- Select 4th neighbors, divide by number of paths to these nodes:
  \[ (1+c) \cdot \frac{t_4}{q} = w \]
- Find 3rd neighbors: \[ Aw \]
- Multiply by paths into 3rd neighbors and tally:
  \[ c += Aw \cdot (q \cdot t_3) \]
Betweenness Centrality: Roll back & Tally

- Select 3rd neighbors, divide by number of paths to these nodes:

\[(1+c) \cdot t_3/q = w\]
Betweenness Centrality: Roll back & Tally

- Select 3rd neighbors, divide by number of paths to these nodes:
  \[(1+c).t_3/q = w\]
- Find 2nd neighbors: \(A\)w

\[A \quad (1+\tilde{c}).t_3/q = Aw\]
Betweenessness Centrality: Roll back & Tally

\[
A \cdot (1+c) \cdot t_3 / q = \text{Aw} \cdot (q \cdot t_2)
\]

- Select 3rd neighbors, divide by number of paths to these nodes:
\[
(1+c) \cdot t_3 / q = w
\]
- Find 2nd neighbors: Aw
- Multiply by paths into 2nd neighbors and tally: \( \tilde{c} += \text{Aw} \cdot (q \cdot t_2) \)
Backward sweep of BC in GraphBLAS C API

```c
...
GrB_Monoid FP32Add; // Monoid <float,+,0.0>
GrB_Monoid_new(&FP32Add,GrB_FP32,GrB_PLUS_FP32,0.0f);
GrB_Monoid FP32Mul; // Monoid <float,*,1.0>
GrB_Monoid_new(&FP32Mul,GrB_FP32,GrB_TIMES_FP32,1.0f);
GrB_Semiring FP32AddMul; // Semiring <float,float,float,+,*,0.0>
GrB_Semiring_new(&FP32AddMul,FP32Add,GrB_TIMES_FP32);

GrB_Monoid nspinv; // inverse of the number of shortest paths
GrB_Monoid_new(&nspinv,GrB_FP32,n,nsver);
GrB_apply(&nspinv,GrB_NULL,GrB_NULL,GrB_MINV_FP32,numsp,GrB_NULL); // nspinv = 1./numsp

GrB_Monoid bcu; // BC updates for each starting vertex in s
GrB_Monoid_new(&bcu,GrB_FP32,n,nsver);
GrB_assign(&bcu,GrB_NULL,GrB_NULL,1.0f,GrB_ALL,n, GrB_ALL,nsver,GrB_NULL); // bcu is filled with 1 to avoid sparsity issues

GrB_Descriptor desc_r; // Descriptor for 1st ewisemult in tally
GrB_Descriptor_new(&desc_r);
GrB_Descriptor_set(desc_r,GrB_OUTP,GrB_REPLACE);
// clear output before result is stored in it.

GrB_Monoid w; // temporary workspace matrix
GrB_Monoid_new(&w,GrB_FP32,n,nsver);
...
```
Backward sweep of BC in GraphBLAS C API

... for (int i=d-1; i>0; i--)
{ // --------------- Tally phase (backward sweep) ---------------

    GrB_eWiseMult(&w,sigmas[i],GrB_NULL,FP32Mul,bcu,nspinv,desc_r);
    // w<sigmas[i]>(1 ./ nsp).*bcu

    // add contributions by successors and mask with that BFS level's frontier
    GrB_mxm(&w,sigmas[i-1],GrB_NULL,FP32AddMul,A,w,desc_r);  // w<sigmas[i-1]> = (A .* w)
    GrB_eWiseMult(&bcu,GrB_NULL,GrB_PLUS_FP32,FP32Mul,w,numsp,GrB_NULL);
    // bcu += w .* numsp
}
// subtract "nsver" from every entry in delta (1 extra value per bcu element crept in)
GrB_assign(delta,GrB_NULL,GrB_NULL, -(float)nsver,GrB_ALL,n,GrB_NULL);  // fill with -nsver
GrB_reduce(delta,GrB_NULL,GrB_PLUS_FP32,GrB_PLUS_FP32,bcu,GrB_NULL);  // add all updates to -nsver

for(int i=0; i<d; i++) { GrB_free(sigmas[i]); }
free(sigmas);
GrB_free_all(frontier,numsp,nspinv,w,bcu,desc_tsr,desc_r);
// macro that expands GrB_free() for each parameter
GrB_free_all(Int32AddMul,Int32Add,FP32AddMul,FP32Add,FP32Mul);
return GrB_SUCCESS;
}
Backward sweep of BC in GraphBLAS C API

```
for (int i=d-1; i>0; i--)
{  // ------------------------ Tally phase (backward sweep) ------------------------
    GrB_eWiseMult(&w,sigmas[i],GrB_NULL,FP32Mul,bcu,nspinv,desc_r);
    // w<sigmas[i]>=(1 ./ nsp).*bcu

    // add contributions by successors and mask with that BFS level's frontier
    GrB_mxmx(&w,sigmas[i-1],GrB_NULL,FP32AddMul,A,w,desc_r);  // w<sigmas[i-1]> = (A +.* w)

    GrB_eWiseMult(&bcu,GrB_NULL,GrB_PLUS_FP32,FP32Mul,w,numsp,GrB_NULL);
    // bcu += w .* numsp
}
// subtract "nsver" from every entry in delta (1 extra value per bcu element created)
GrB_assign(delta,GrB_NULL,GrB_NULL,-((float)nsver),GrB_ALL,n,GrB_NULL);
// add all updates to -nsver
GrB_reduce(delta,GrB_NULL,GrB_PLUS_FP32,GrB_PLUS_FP32,bcu,GrB_NULL);

for(int i=0; i<d; i++) { GrB_free(sigmas[i]); }  
free(sigmas);
GrB_free_all(frontier,numsp,nspinv,w,bcu,desc_tsr,desc_r);
// macro that expands GrB_free() for each parameter
GrB_free_all(Int32AddMul,Int32Add,FP32AddMul,FP32Add,FP32Mul);
return GrB_SUCCESS;
```

- The contributions of each “end” vertex to its predecessors are divided by the number of shortest paths that reach them.
- This is accomplished with an eWiseMult operation where the sigma[i] matrix is used as a mask to ensure that only paths identified in the BFS phase (i.e. edges that belong to the BFS tree) are assigned to the result.
Backward sweep of BC in GraphBLAS C API

... for (int i=d-1; i>0; i--)
{ // ---------------------- Tally phase (backward sweep) ----------------------
   GrB_eWiseMult(&w,sigmas[i],GrB_NULL,FP32Mul,bcu,nspinv,desc_r);
   // w<sigmas[i]>(1 ./ nsp).*bcu

   // add contributions by successors and mask with that BFS level's frontier
   GrB_mxnm(&w,sigmas[i-1],GrB_NULL,FP32AddMul,A,w,desc_r);  // w<sigmas[i-1]>(A +.* w)

   GrB_eWiseMult(&bcu,GrB_NULL,GrB_PLUS_FP32,FP32Mul,w,numsp,GrB_NULL);
   // bcu += w .* numsp
}
// subtract "nsver" from every entry in delta (1 extra value per bcu element crept in)
GrB_assign(delta,GrB_NULL,GrB_NULL,(float)-nsver,GrB_NULL)
GrB_reduce(delta,GrB_NULL,GrB_PLUS_FP32,GrB_PLUS_FP32,bcu,GrB_NULL);
// add all updates

for(int i=0; i<d; i++)
   free(sigmas);
free(sigmas);
GrB_free_all(
   Int32AddMul,Int32Add,FP32AddMul,FP32Add,FP32Mul,
   GrB_free_all(0)
)
return GrB_SUCCESS;

- The GrB_mxnm call discovers predecessors (as opposed to successors in the forward sweep) by its use of the descriptor desc_r that uses the adjacency matrix (as opposed to its transpose).
- The algorithm assures that the BC contributions are transferred only to direct parents on the BFS tree by passing the previous level of BFS tree (sigma[i-1]) as a mask to GrB_mxnm.