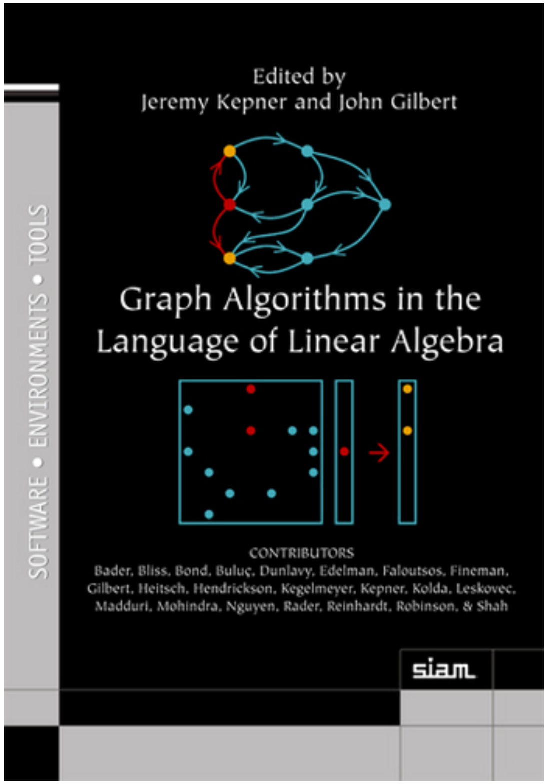
Concepts in GraphBLAS C API

Aydın Buluç
Berkeley Lab (LBNL)

Joint work with Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang
March 9, 2017

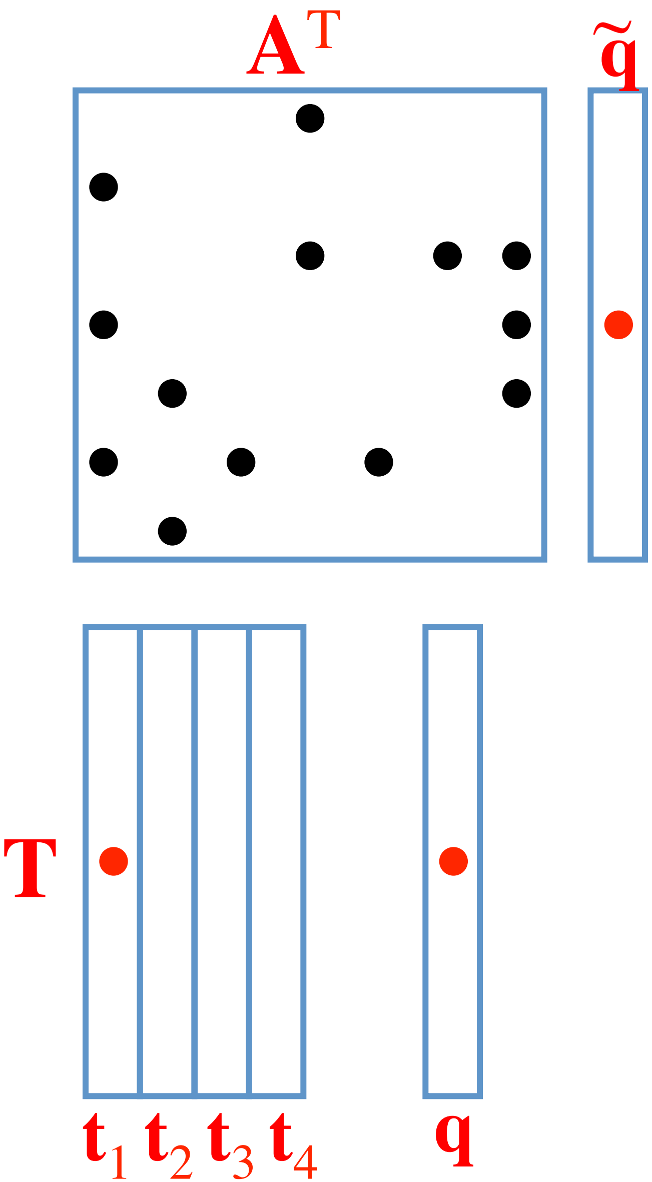
Fast-forward in history

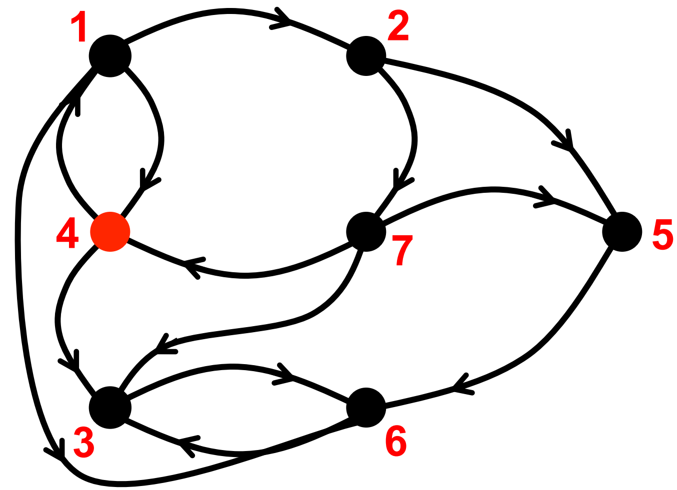
- The idea is older than this SIAM book:
- Several platforms implemented the ideas in the past, such as Star*P
- Current list of active implementations (and a small portion of the draft proposal) is available at http://graphblas.org
- Let's start with an example: **Betweenness Centrality**



Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, Carl Yang. "Proposal for a GraphBLAS C API" (Working document from the GraphBLAS Signatures Subgroup)

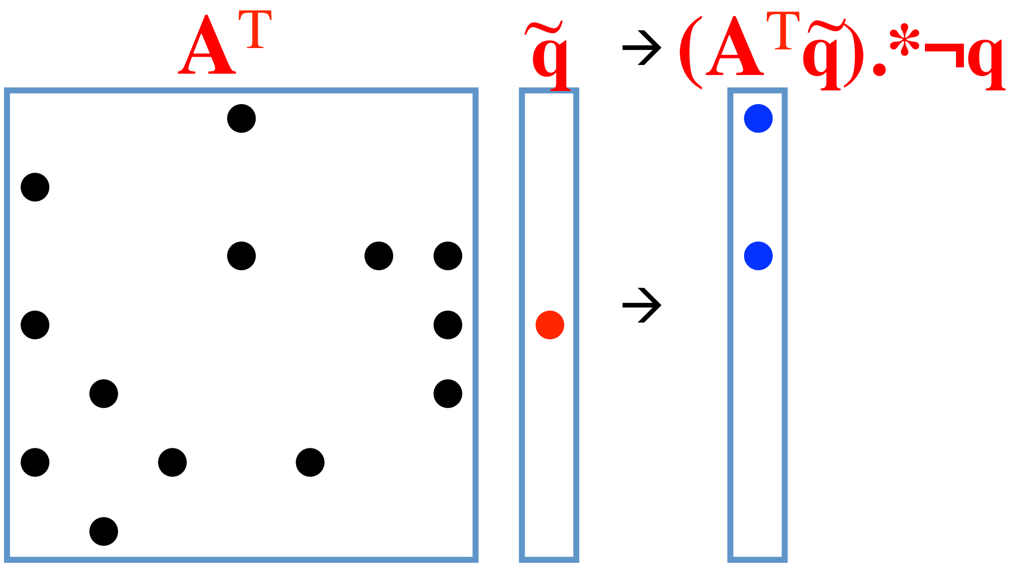
Betweenness Centrality: Data Structures

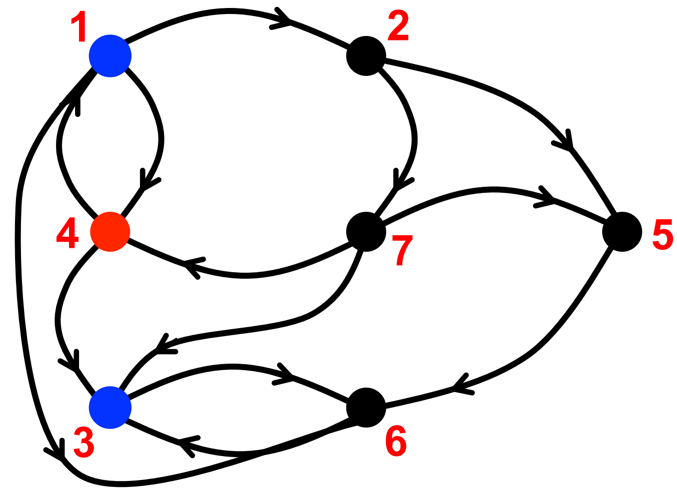


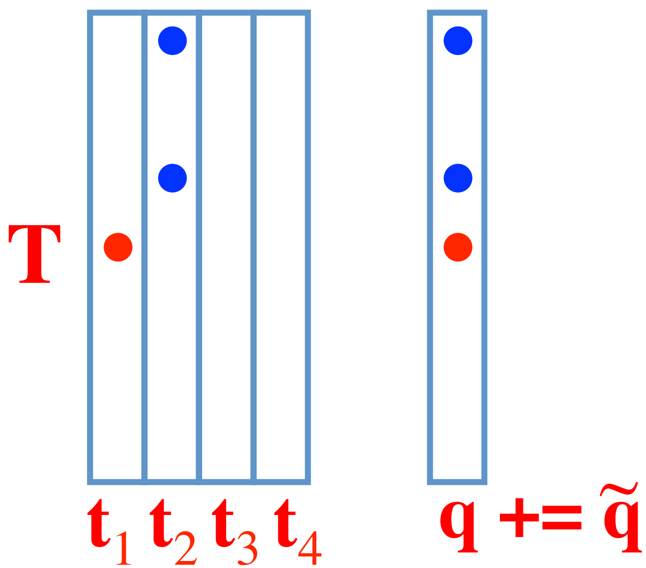


- Pick a starting vertex (4)
- Initialize vectors: q, $\tilde{q}$, and t_d

Betweenness Centrality: Get Neighbors

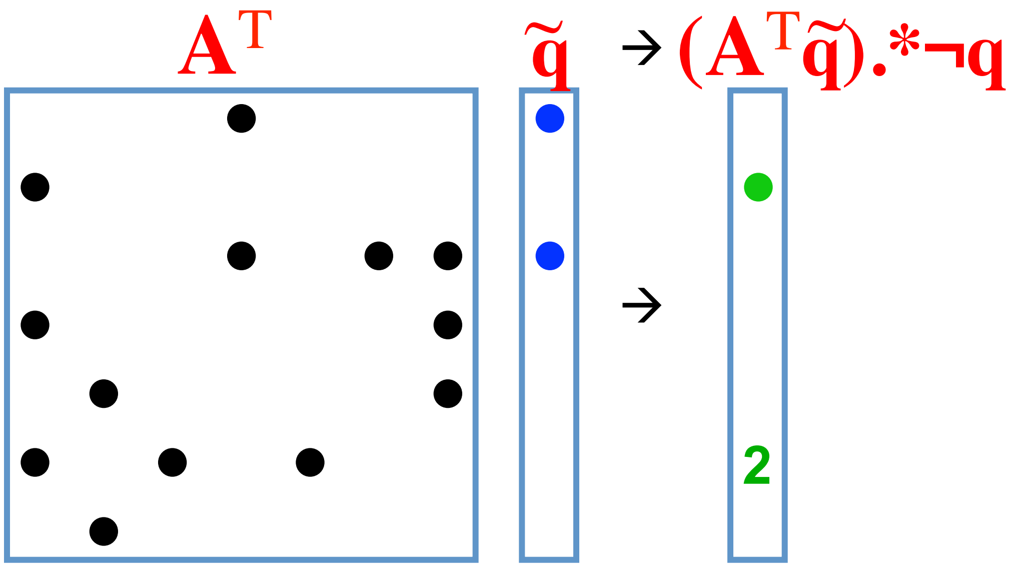


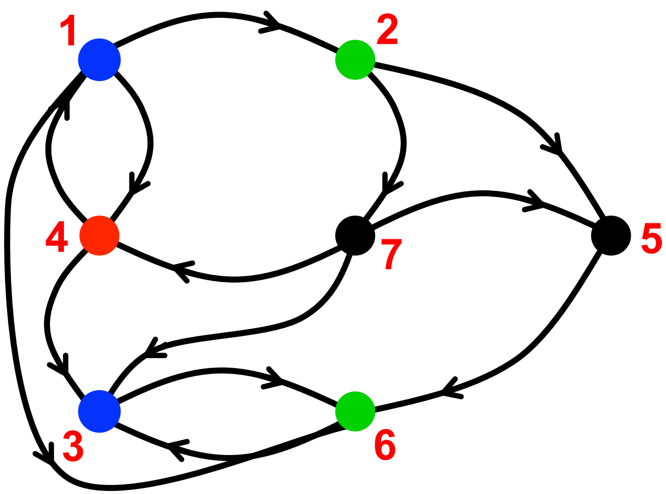


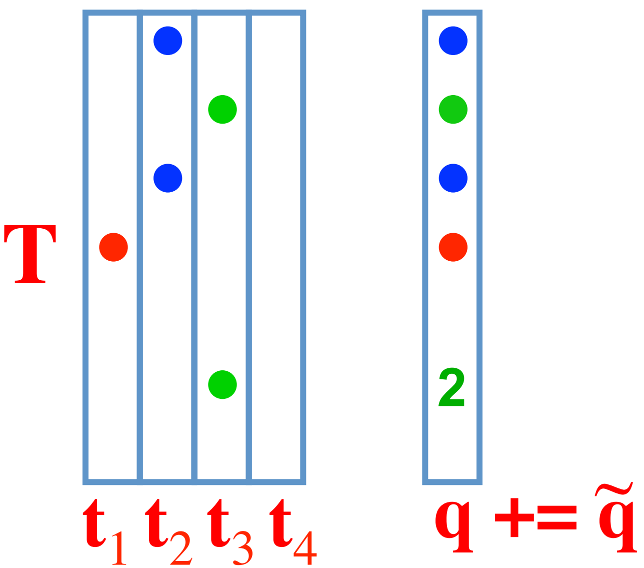


- Get 2nd neighbors from starting vertex: $A^T\tilde{q}$
- Eliminate existing vertices: $.{*}\neg q$
- Tally: $q \mathrel{+}= \tilde{q}$
- Update table: $t_2 = \tilde{q}$

Betweenness Centrality: Get Neighbors

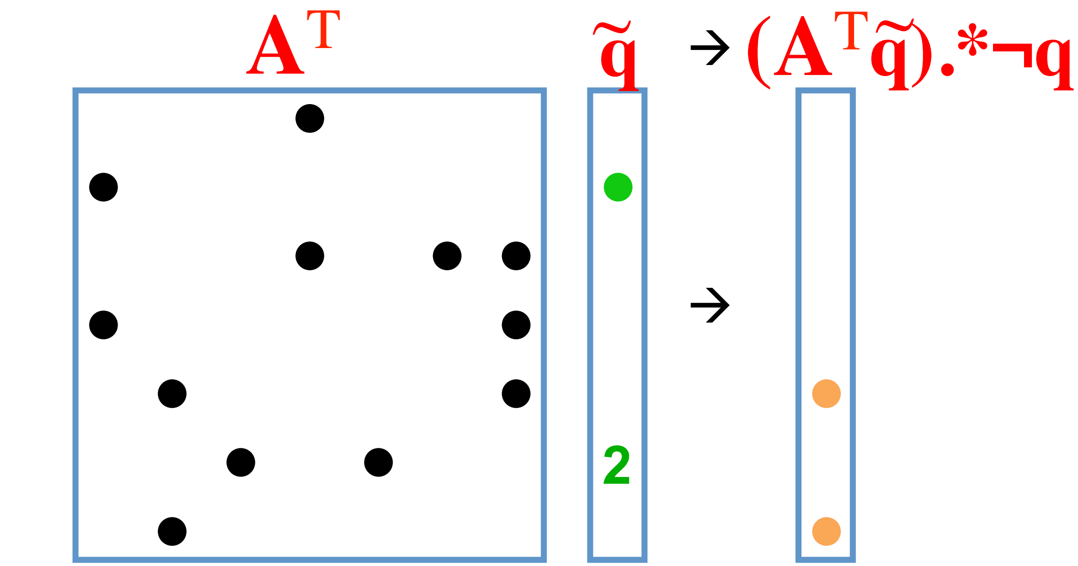


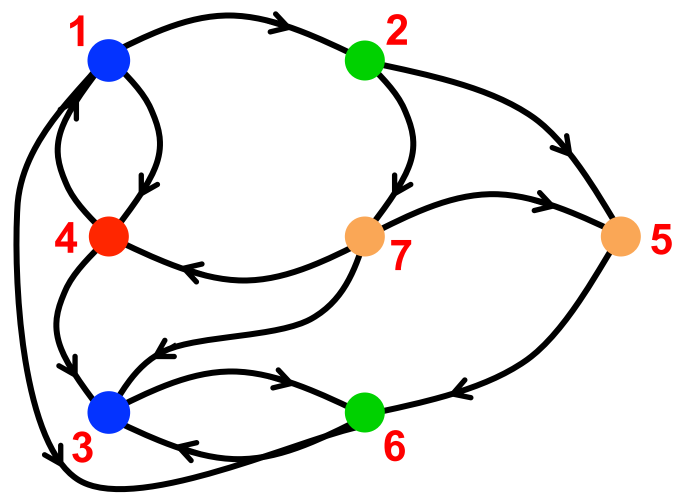


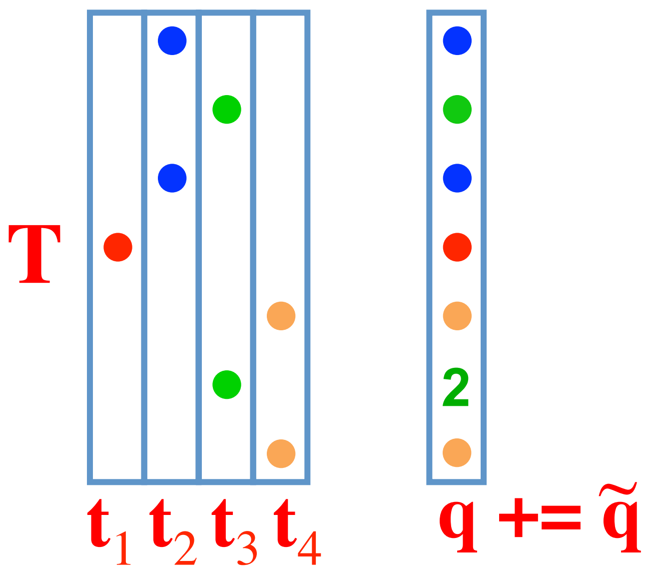


- Get 3rd neighbors from starting vertex: $A^T\tilde{q}$; sum paths to vertex
- Eliminate existing vertices: $.*\neg q$
- Tally: $q+=\tilde{q}$
- Update table: $t_2 = \tilde{q}$

Betweenness Centrality: Get Neighbors

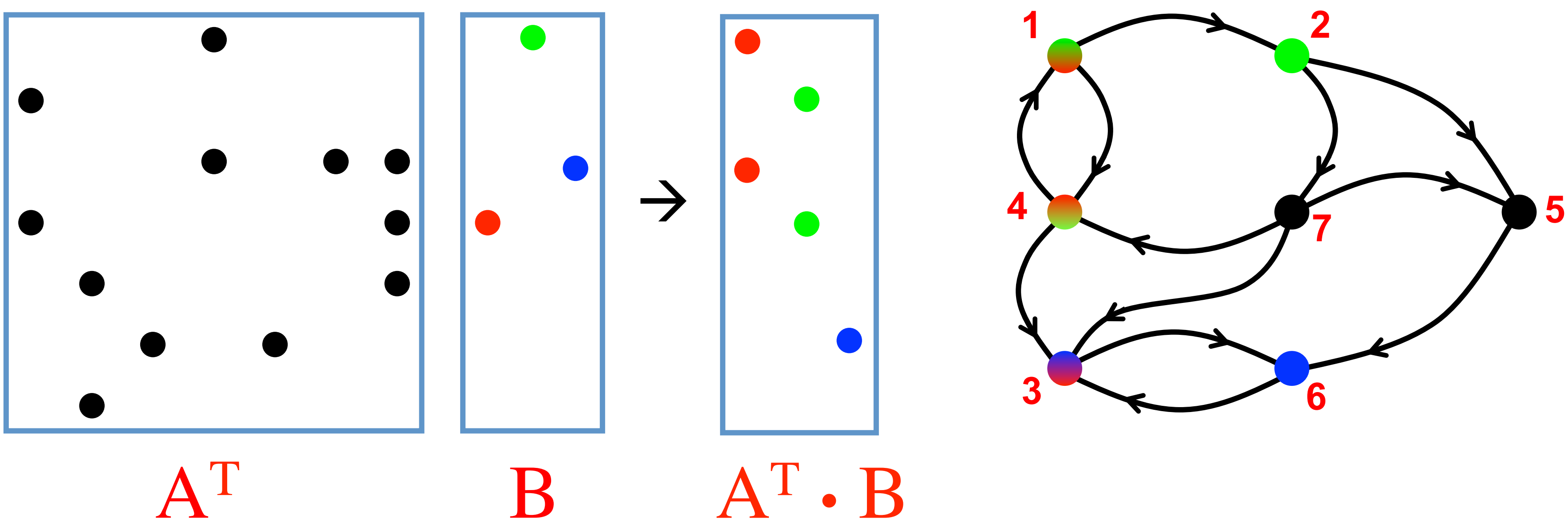






- Get 4th neighbors from starting vertex: $A^T\tilde{q}$
- Eliminate existing vertices: $.*\neg q$
- Tally: $q \mathrel{+}= \tilde{q}$
- Update table: $t_2 = \tilde{q}$

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

Fig. 2: The GrB_mxm() function signature, parameters, and return values.

a) 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);
```

b) Parameters:

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: $\mathsf{op} = \langle D_1, D_2, D_3, \oplus, \otimes, 0\rangle$.

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.

c) Return Values:

GrB_SUCCESS	blocking mode: the operation completed successfully. Nonblocking mode: consistency tests passed on dimensions and domains for the input arguments.
GrB_PANIC	Unknown internal error
GrB_OUTOFMEM	Not enough memory available for operation
GrB_DIMENSION_MISMATCH	Matrix dimensions are incompatible.
GrB_DOMAIN_MISMATCH	The domains of the various matrices are incompatible with the corresponding domains of the accumulating operation, semiring, or mask.

Algebraic structures in GraphBLAS

Functions: $F = \langle D1, D2, D3, \oplus \rangle$ is defined by three domains D1, D2, D3, and an operation $\oplus : D1 \times D2 \rightarrow D3$

Monoids: $M = \langle D1, D2, D3, \oplus, 0 \rangle$ is defined by three domains D1, D2, D3, an operation $\oplus : D1 \times D2 \rightarrow D3$, and an element $0 \in D3$

Semirings: $S = \langle D1, D2, D3, \oplus, \otimes, 0[, 1] \rangle$ is defined by three domains D1, D2 and D3, an additive operation $\oplus : D3 \times D3 \rightarrow D3$, a multiplicative operation $\otimes : D1 \times D2 \rightarrow D3$, an element $0 \in D3$ and an optional element $1 \in D3$

In the special case of $D1 = D2 = D3$, 1 defined, and 0 working as the $\otimes$ annihilator (i.e., $0 \otimes x = x \otimes 0 = 0, \forall x \in D3$), a GraphBLAS semiring ***reduces to the conventional semiring algebraic structure***

Objects in GraphBLAS

Vectors: A vector $v = \langle D, N, \{(i, v_i)\} \rangle$ is defined by a domain D, a size $N > 0$ and a set of tuples (i, v_i) where $0 \leq i < N$ and $v_i \in D$.

Matrices: A matrix $A = \langle D, M, N, \{(i, j, A_{ij})\} \rangle$ is defined by a domain D. its number of rows $M > 0$, its number of columns $N > 0$ and a set of tuples (i, j, A_{ij}) where $0 \leq i < M$, $0 \leq j < N$, and $A_{ij} \in D$.

Descriptors: Descriptors are used as input parameters in various GraphBLAS methods to provide more details of the operation to be performed by those methods.

Forward sweep of BC in GraphBLAS C API

```
#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( , GrB_INT32, n, nsver); // also stores # of shortest paths so far

  GrB_Matrix_build( ,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( ,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

…

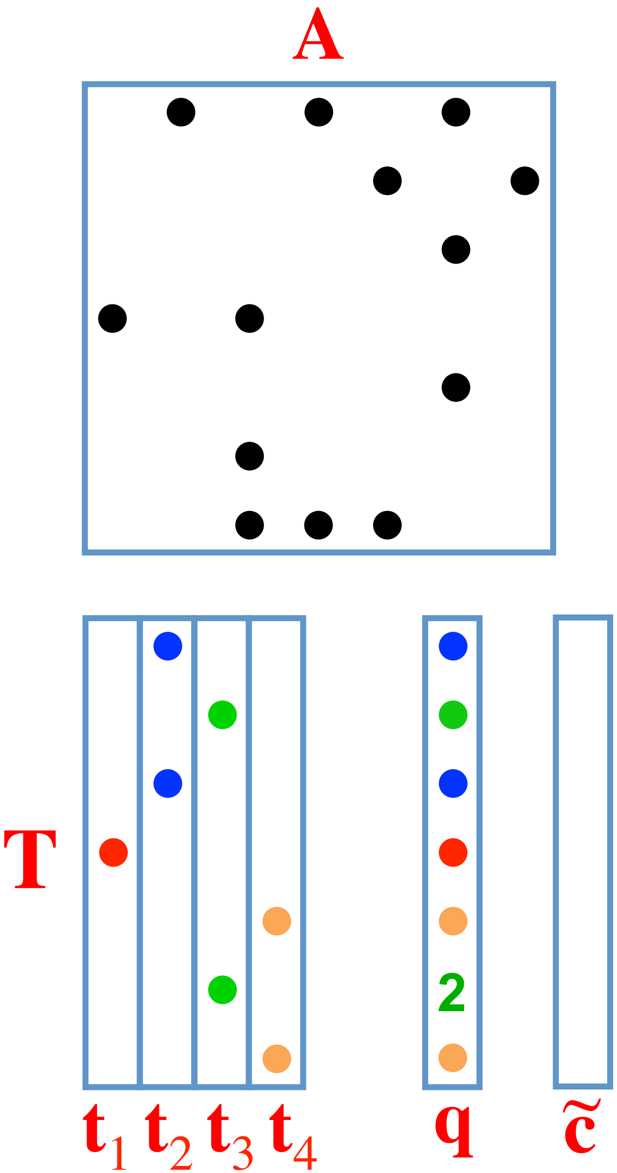
```
GrB_Matrix nu
GrB_Matrix_ne

GrB_Matrix_bu
free(i_nsver)

GrB_Matrix fr
GrB_Matrix_ne
GrB_extract(&

// The memory
GrB_Matrix *s
int32_t d = 0
int32_t nvals
do {  // ----
  GrB_Matrix_
  // sigmas[d

  GrB_apply(&
  GrB_eWiseAd
  // 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);
```

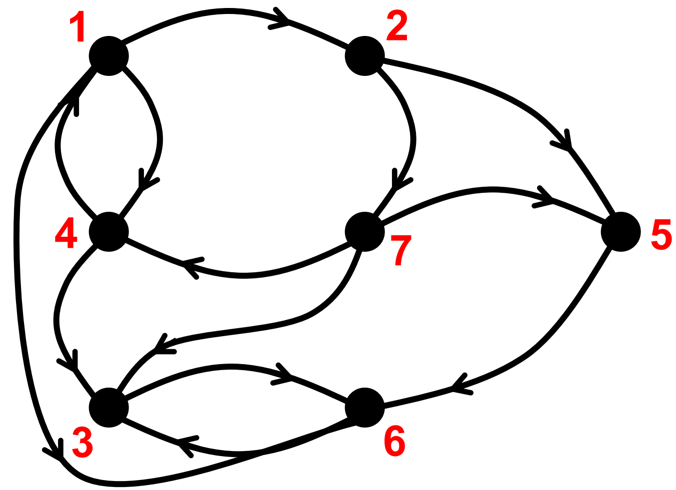
…

- 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.

- Introduction
- Array Approach
  - *BC: Data Structures*
  - *BC: Shortest Paths*
  - ***BC: Rollback & Tally***
- Array Algorithm
- Results
- Summary

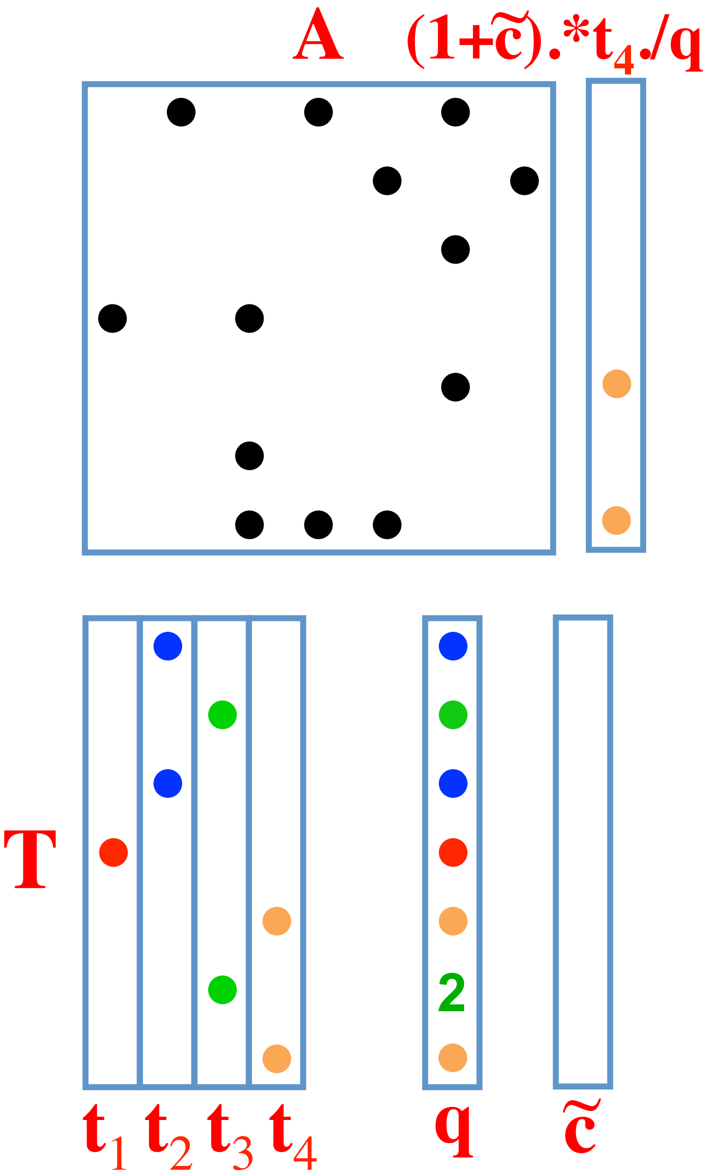
Betweenness Centrality: Roll back & Tally

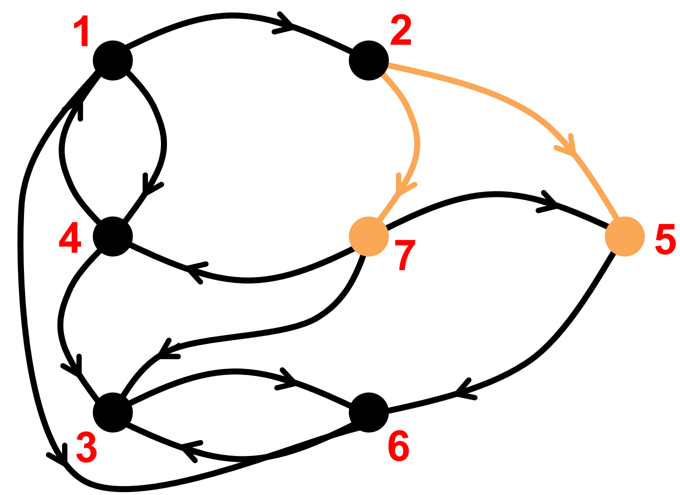




- **Initialize the centrality update: $\tilde{c}$**
- **Will hold the contributions of these shortest paths to each vertexes betweenness centrality**

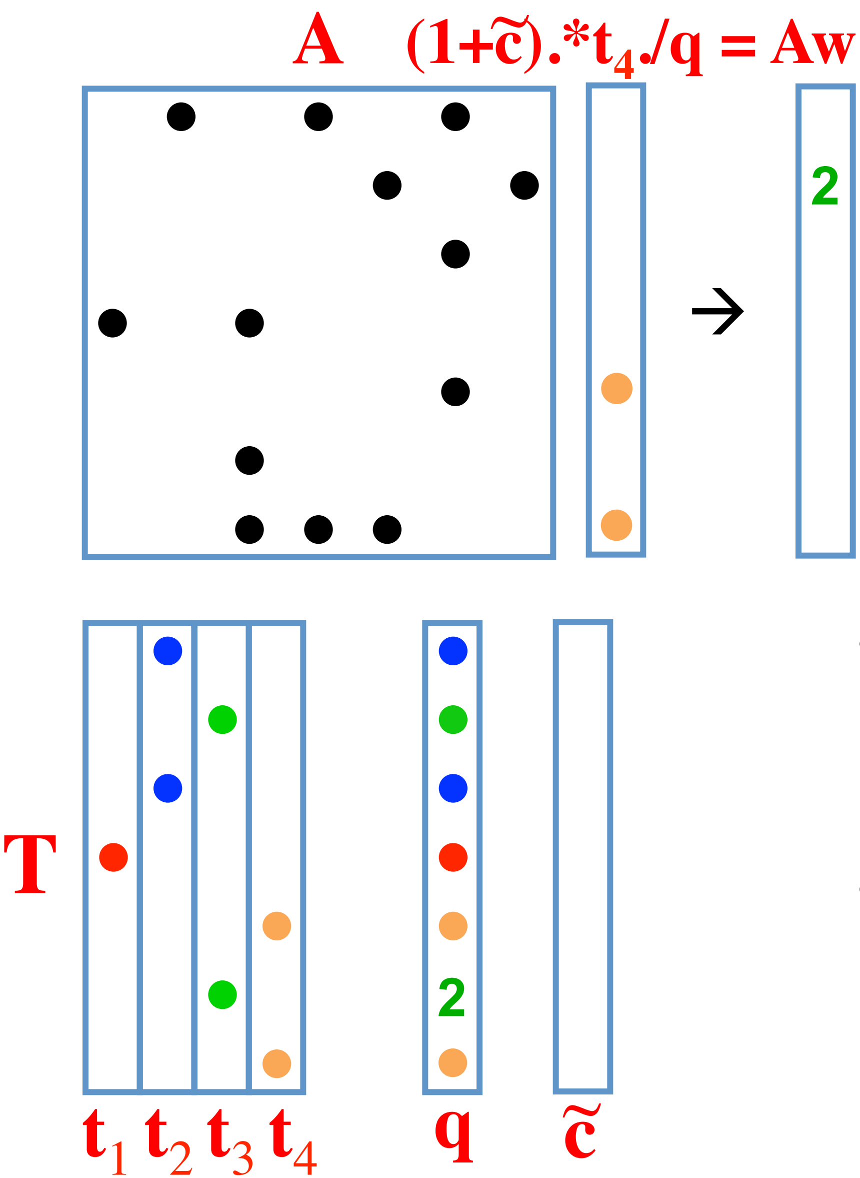
Betweenness Centrality: Roll back & Tally

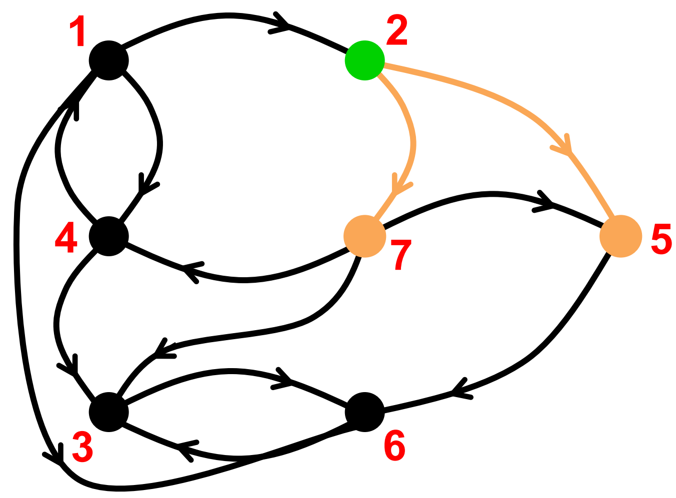




- Select 4th neighbors, divide by number of paths to these nodes: $(1+c).*t_4./q = w$

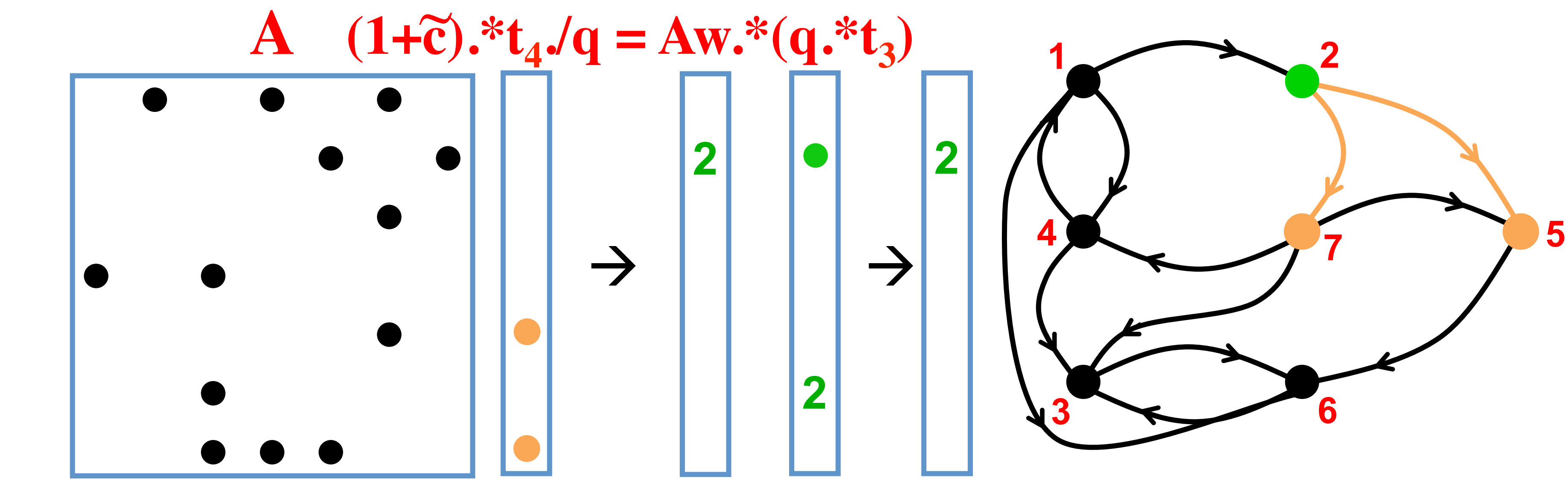
Betweenness Centrality: Roll back & Tally

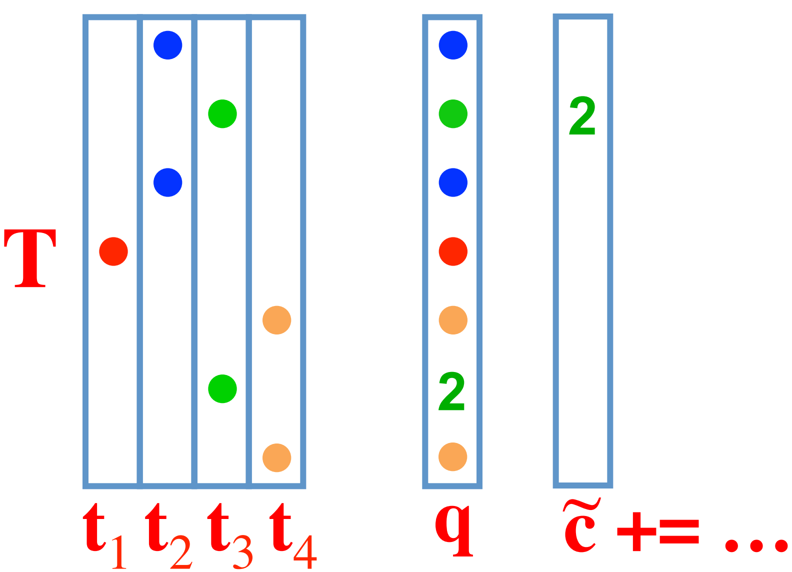




- Select 4th neighbors, divide by number of paths to these nodes: $(1+c).*t_4./q = w$
- Find 3rd neighbors: Aw

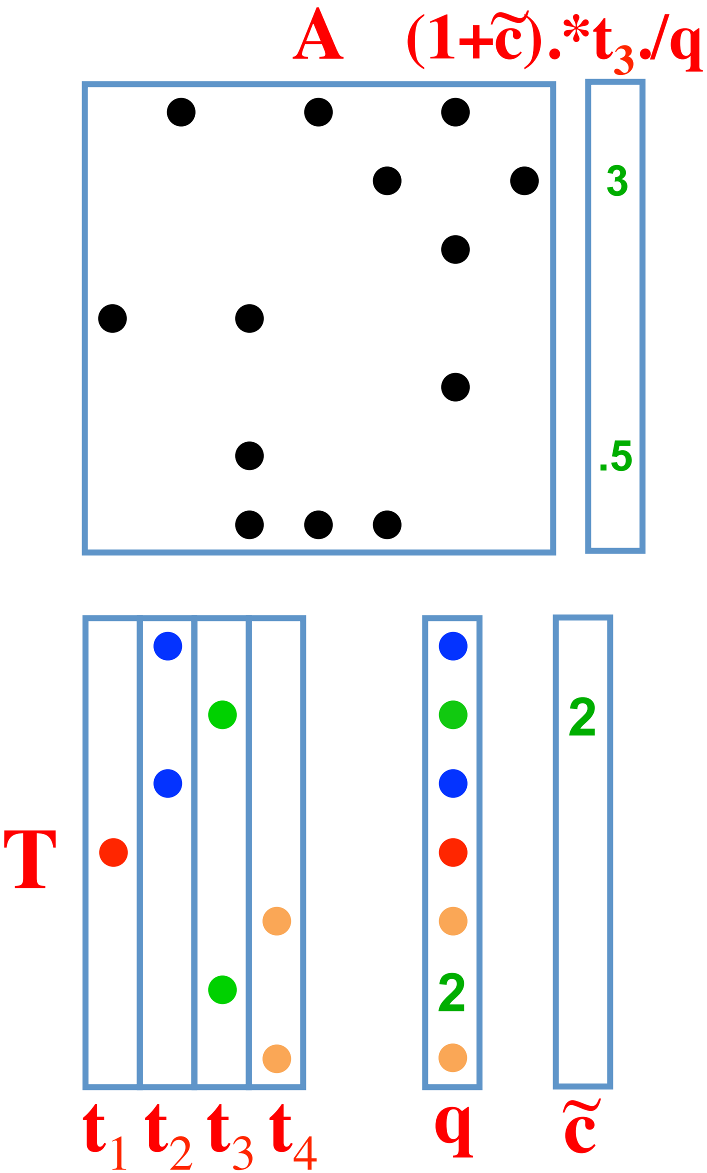
Betweenness Centrality: Roll back & Tally

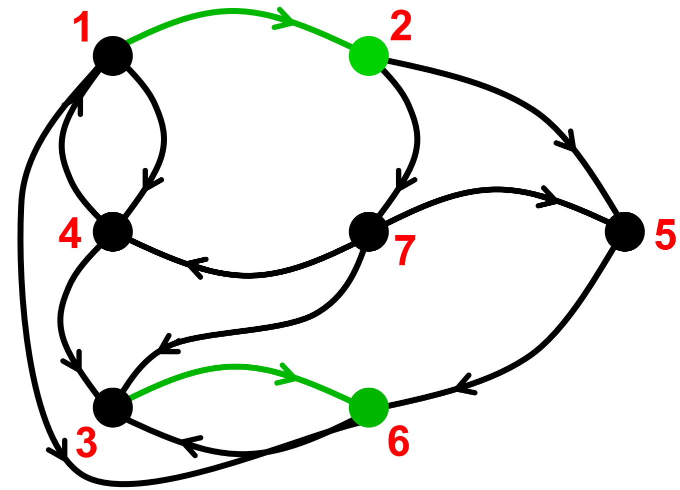




- Select 4th neighbors, divide by number of paths to these nodes: $(1+c).*t_4./q = w$
- Find 3rd neighbors: Aw
- Multiply by paths into 3rd neighbors and tally: $c\mathrel{+}=Aw.*(q.*t_3)$

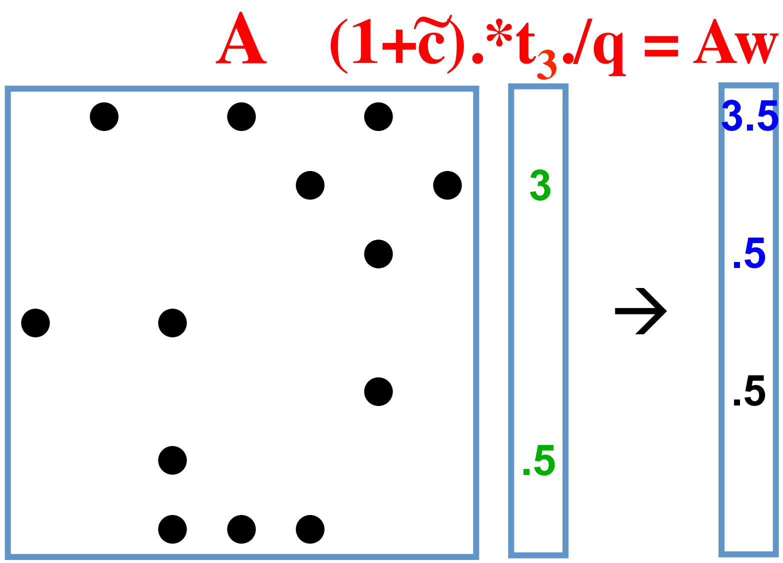
Betweenness Centrality: Roll back & Tally

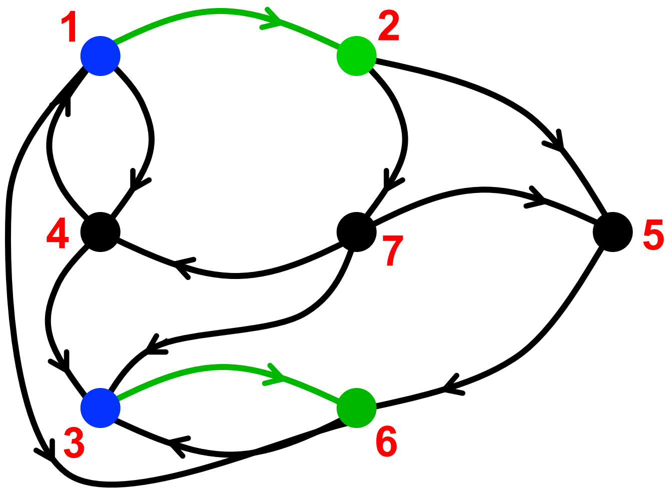


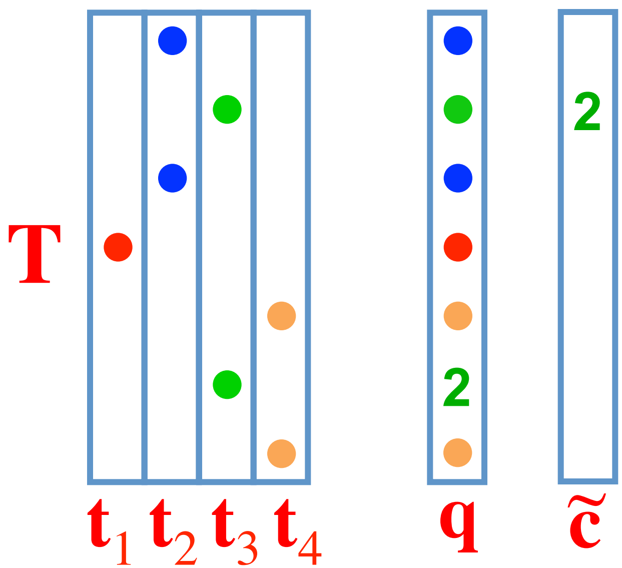


- Select 3rd neighbors, divide by number of paths to these nodes: $(1+c).*t_3./q = w$

Betweenness Centrality: Roll back & Tally

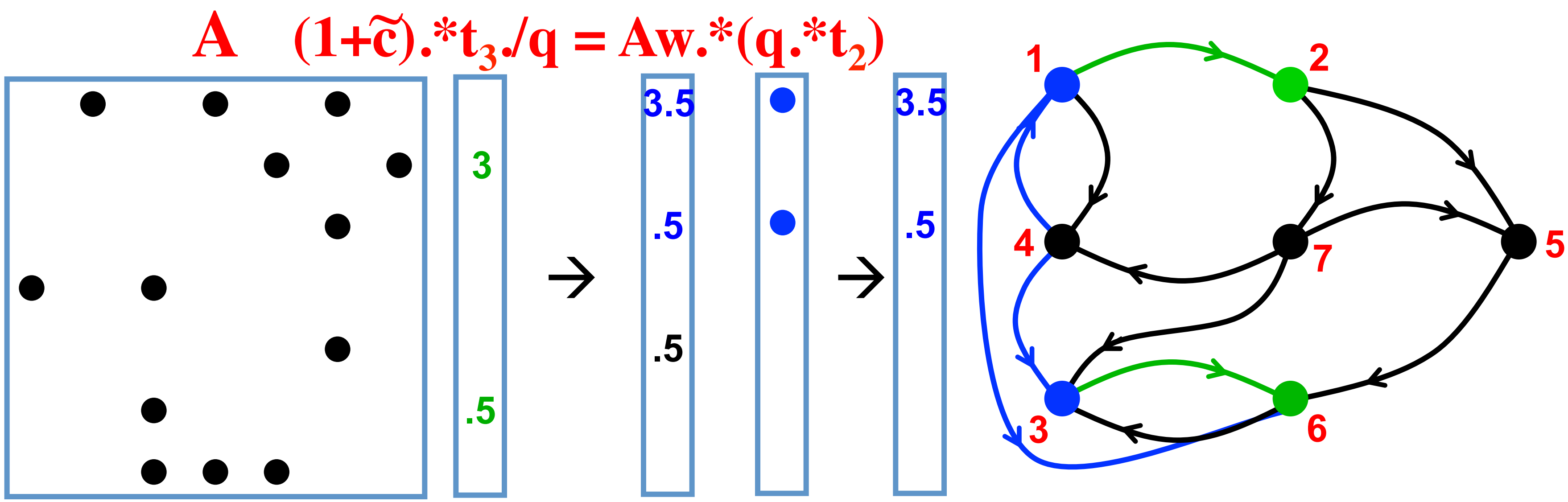


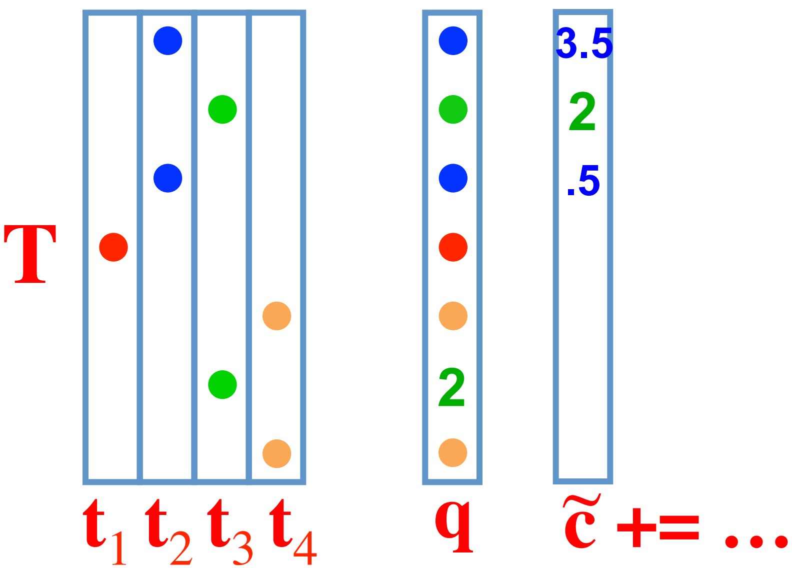




- Select 3rd neighbors, divide by number of paths to these nodes: $(1+c).*t_3./q = w$
- Find 2nd neighbors: Aw

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: Aw
- Multiply by paths into 2nd neighbors and tally: $\tilde{c}$+=$Aw.*(q.*t_2)$

Backward sweep of BC in GraphBLAS C API

```
…
  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_Matrix nspinv;   // inverse of the number of shortest paths
  GrB_Matrix_new(&nspinv,GrB_FP32,n,nsver);
  GrB_apply(&nspinv,GrB_NULL,GrB_NULL,GrB_MINV_FP32,numsp,GrB_NULL);  // nspinv = 1./numsp

  GrB_Matrix bcu;   // BC updates for each starting vertex in s
  GrB_Matrix_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_Matrix w;                                             // temporary workspace matrix
  GrB_Matrix_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_mxm(&w,sigmas[i-1],GrB_NULL,FP32AddMul,A,w,desc_r);   // w<sigmas[i-1]> = (A +.* w)

    GrB_eW
    // bcu
  }
  // subtr                                                                  )
  GrB_assi                                                                  -nsver
  GrB_redu
  // add a

  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_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(de
  GrB_reduce(de
  // add all up

  for(int i=0;
  free(sigmas);
  GrB_free_all(
  // macro that
  GrB_free_all(
  return GrB_SU
}
```

- The GrB_mxm 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_mxm.

Important Concepts

- Masks avoids computation and materialization of intermediate objects.
- The BC example shows how we both expand the current frontier and prune the previously discovered vertices via a single call to mxm (or vxm) using masks.
- All masks are "write" masks (i.e. they apply to the output as opposed to any intermediate product).
- Any object (not just Boolean) can be passed as a mask
- Check the spec for the intricate semantics of mixing masks and accumulators).
- Sparsity of matrix/vector objects are not declarative (runtime determines storage)
- Mixed-type arithmetic is achieved via either by the user specified semirings or by relying on the type casting abilities of the underlying language.

Common elements in function calls:

- GrB_ "namespace"
- Destination object is the first parameter
- Mask matrix and accumulation function are next (if supported)
    - Pass GrB_NULL if not needed.
- Descriptor is optional and is always last (or use GrB_NULL)

Important Concepts

- All objects are opaque. But: opaque ≠ undefined
- i.e. opaque objects need to be “defined” by the implementation

Supported execution models:

- **blocking:** Each method in a sequence completes the GraphBLAS operation defined by the method before proceeding to the next statement in program order. Output GraphBLAS objects defined by a method are stored in memory and are available to other C functions after each method returns.
- **nonblocking:** Each method may return once the input arguments have been inspected and verified to define a well formed GraphBLAS operation. The GraphBLAS operation and the state of any GraphBLAS objects are undefined when a method returns until the terminating method in the sequence returns.

Nonblocking mode allows for any execution strategy that satisfies the mathematical definition of the sequence. The methods can be placed into a queue and deferred. They can be chained together and fused (e.g. replacing a chained pair of matrix products with a matrix triple product). Lazy evaluation, greedy evaluation or asynchronous execution are all valid as long as the final result agrees with the mathematical definition provided by the sequence of GraphBLAS method calls appearing in program order.