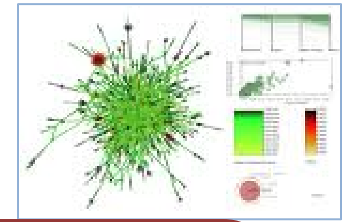
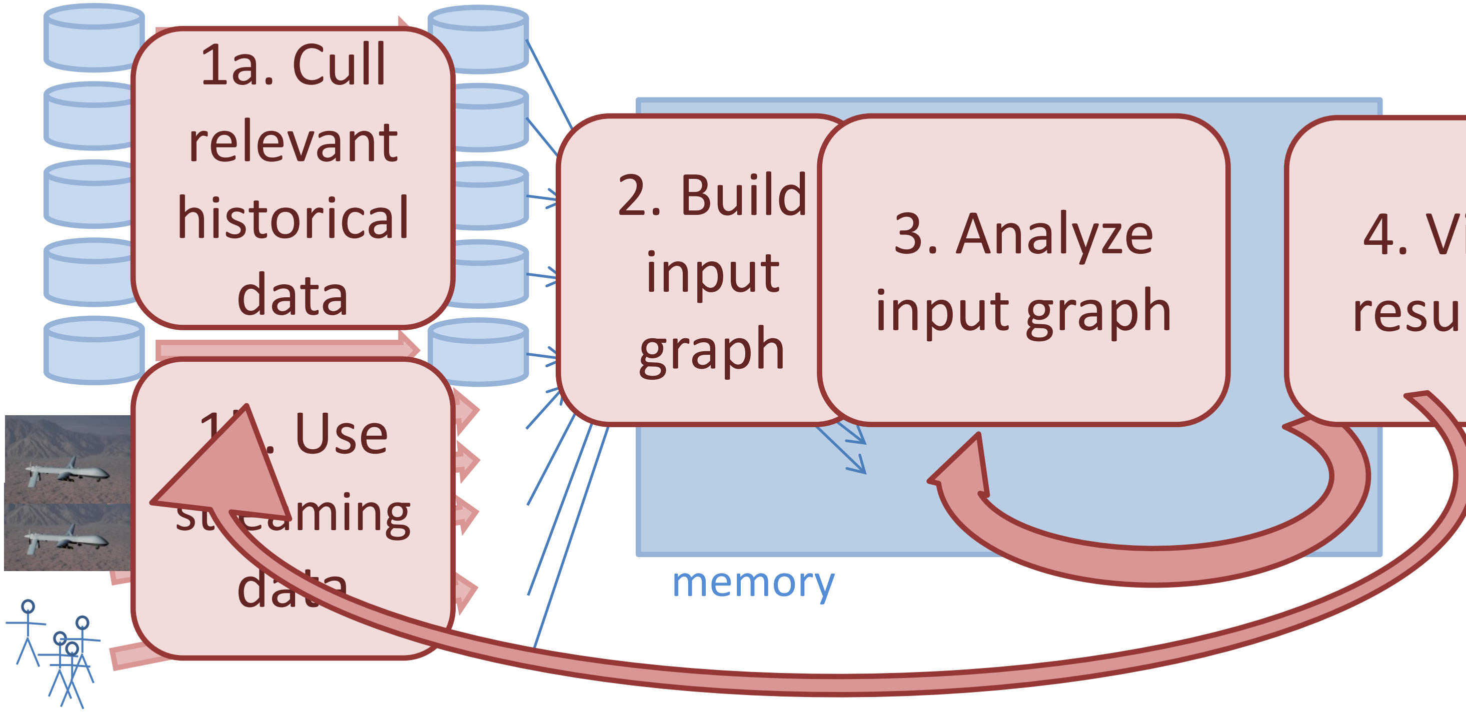
Deployed Large-Scale Graph Analytics: Use Cases, Target Audiences, and Knowledge Discovery Toolbox (KDT) Technology

Aydin Buluc, LBL (abuluc@lbl.gov)
John Gilbert, Adam Lugowski and Drew Waranis, UCSB ({gilbert,alugowski,awaranis}@cs.ucsb.edu)
David Alber and **Steve Reinhardt**, Microsoft ({david.alber,steve.reinhardt}@microsoft.com)

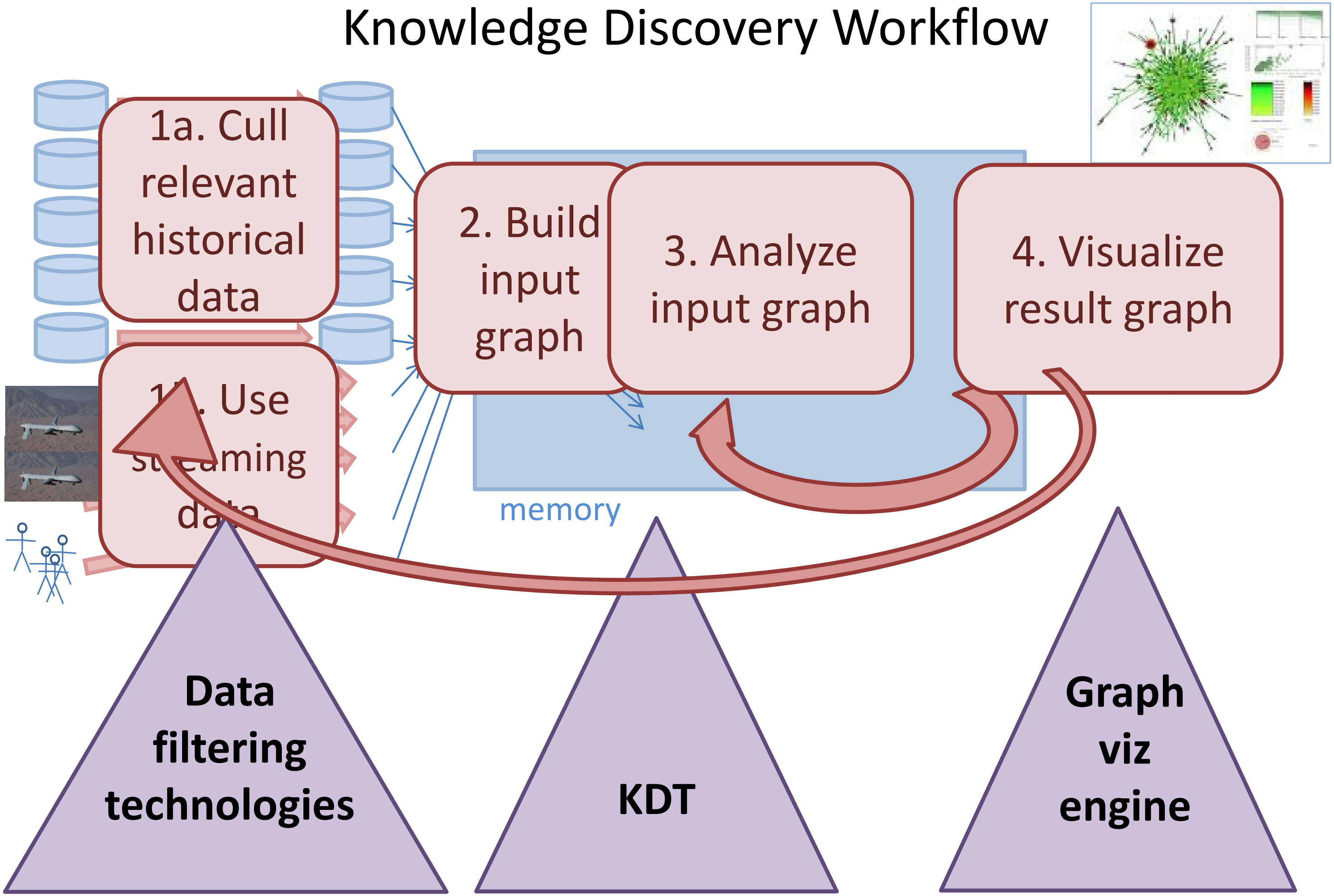
Knowledge Discovery Toolbox enables rapid algorithm development and fast execution for large-scale complex graph analytics

Knowledge Discovery Workflow





Knowledge Discovery Workflow



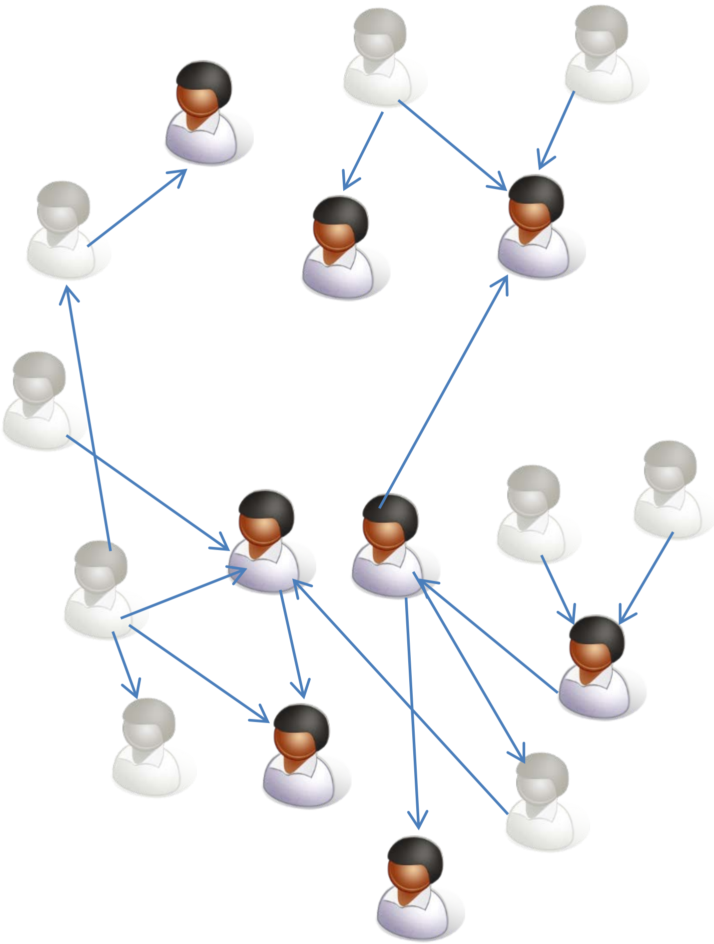
Agenda

- Use cases and audiences for graph analytics
- Technology
- Next steps

Graph Analytics

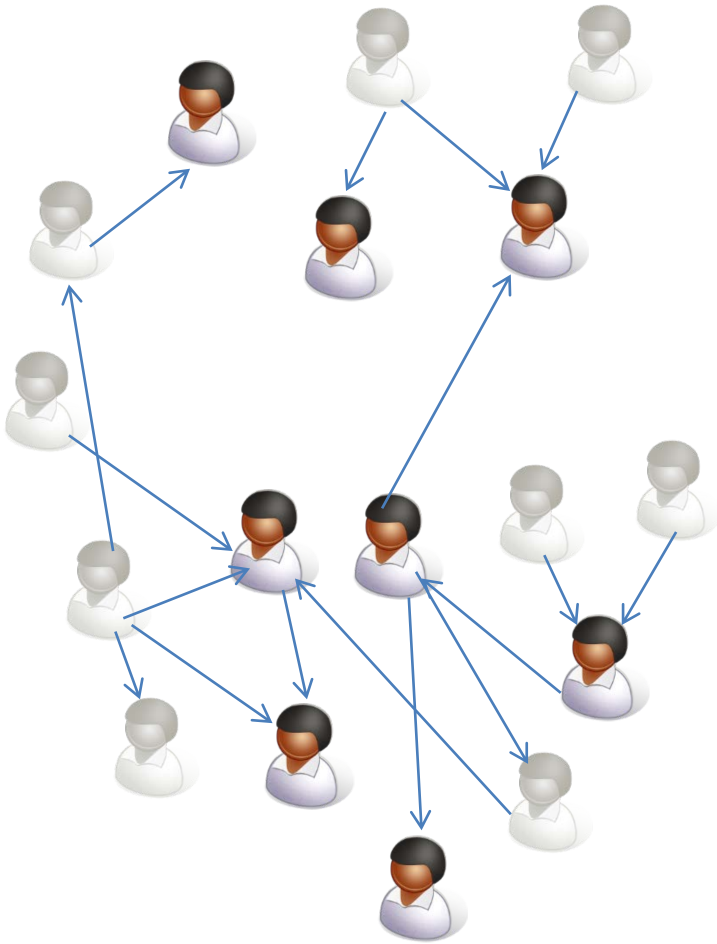
- Graphs arise from
  - Social networks (human or animal)
  - Transaction networks (*e.g.,* Internet, banking)
  - Molecular biological interactions *(e.g.,* protein-protein interactions)
- Many queries are
  - Ranking
  - Clustering
  - Matching / Aligning
- Graphs are not all the same
  - Directed simple graphs, hypergraphs, bipartite graphs, with or without attributes on edges or vertices, …

Use Case: Find Influential People in a Social Network



Use Case: Find Influential People in a Social Network


Warfighters



- Warfighter wants to understand a social network (*e.g.*, village, terrorist group); see DARPA GUARDDOG
- Specifically, wants to identify leaders / influencers
- GUI selects data, calls KDT to identify top N influencers

Use Cases

- Homeland security / Understand roles of members of terrorist groups based on known links between them / “Looking just at cell-phone communications, who are the leaders?”
- International banking / Detect money laundering / “Find instances of money being transferred at least 5 times and coming back to its source.”

Common thread: Enabling the knowledge-discovery domain expert to analyze graphs directly gets to the “right” answer faster and possibly at all. (In the embedded context, the end-user and the KD domain expert are likely different people.)

Audiences

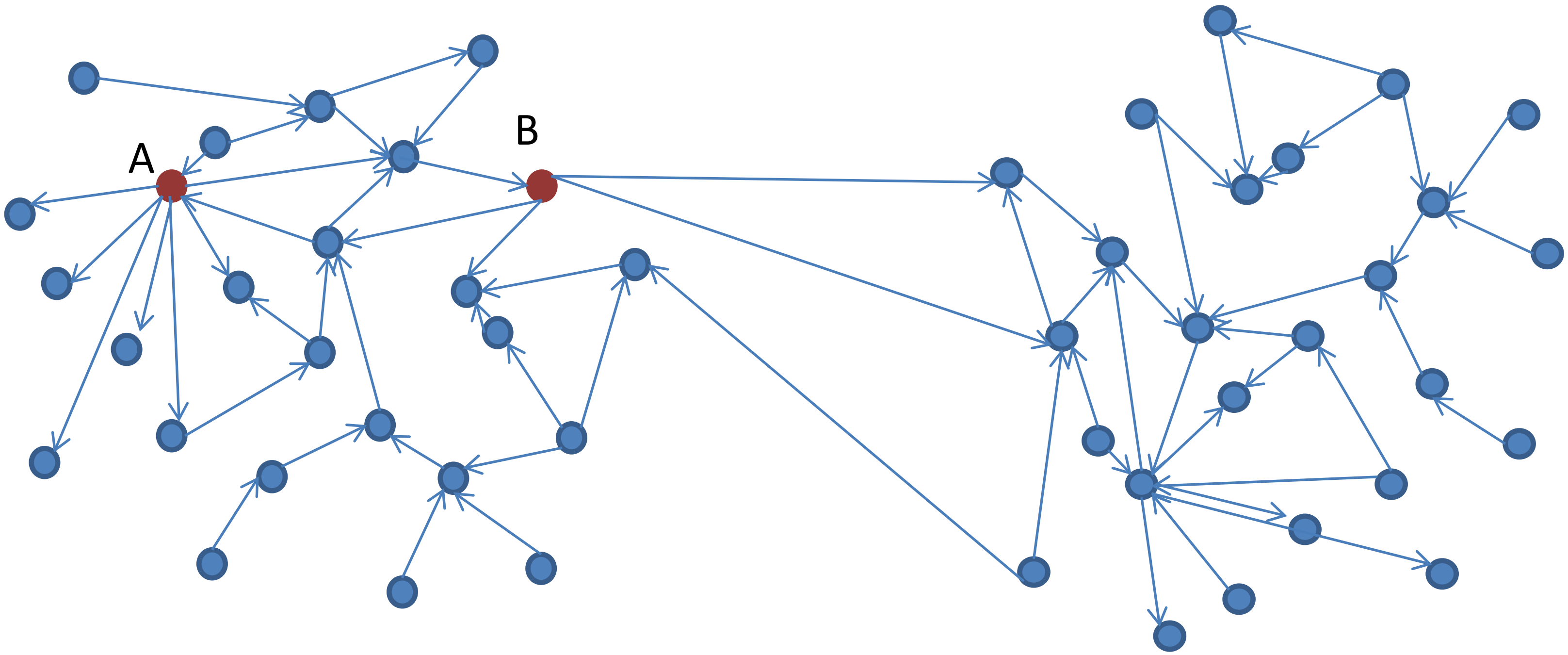
- End-users / warfighters
  - True end-user GUI not addressed by KDT
- Knowledge discovery domain experts
  - Are experts in something other than graph analytics
  - Have large graphs they need to explore as part of their work
  - Want simple, robust, scalable, flexible package
- Graph-analytic researchers
  - Are experts in graph analytics, machine learning, etc.
  - Want to experiment with new algorithms …
  - And get feedback from users on efficacy on large data
- Efficiency-level developers
  - Call-backs in C++ currently have big performance advantage
  - Formatting data for ingest

Agenda

- Use cases and audiences for graph analytics
- Technology
- Next steps

Local v. Global Metrics

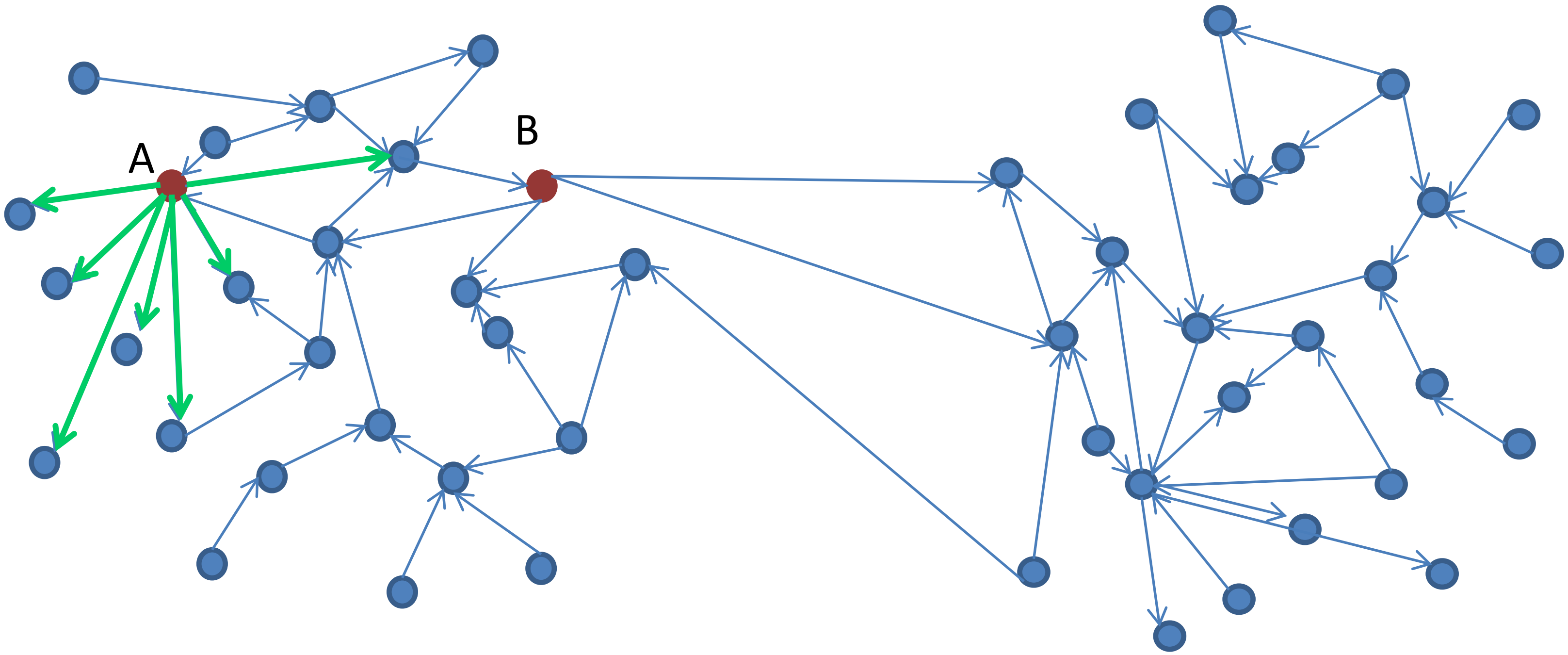
Degree Centrality v. Betweenness Centrality



- Is vertex A or B most central?
  - A has directed edges to more vertices (degree centrality)
  - B is on more shortest paths between vertex pairs (betweenness centrality)

Local v. Global Metrics

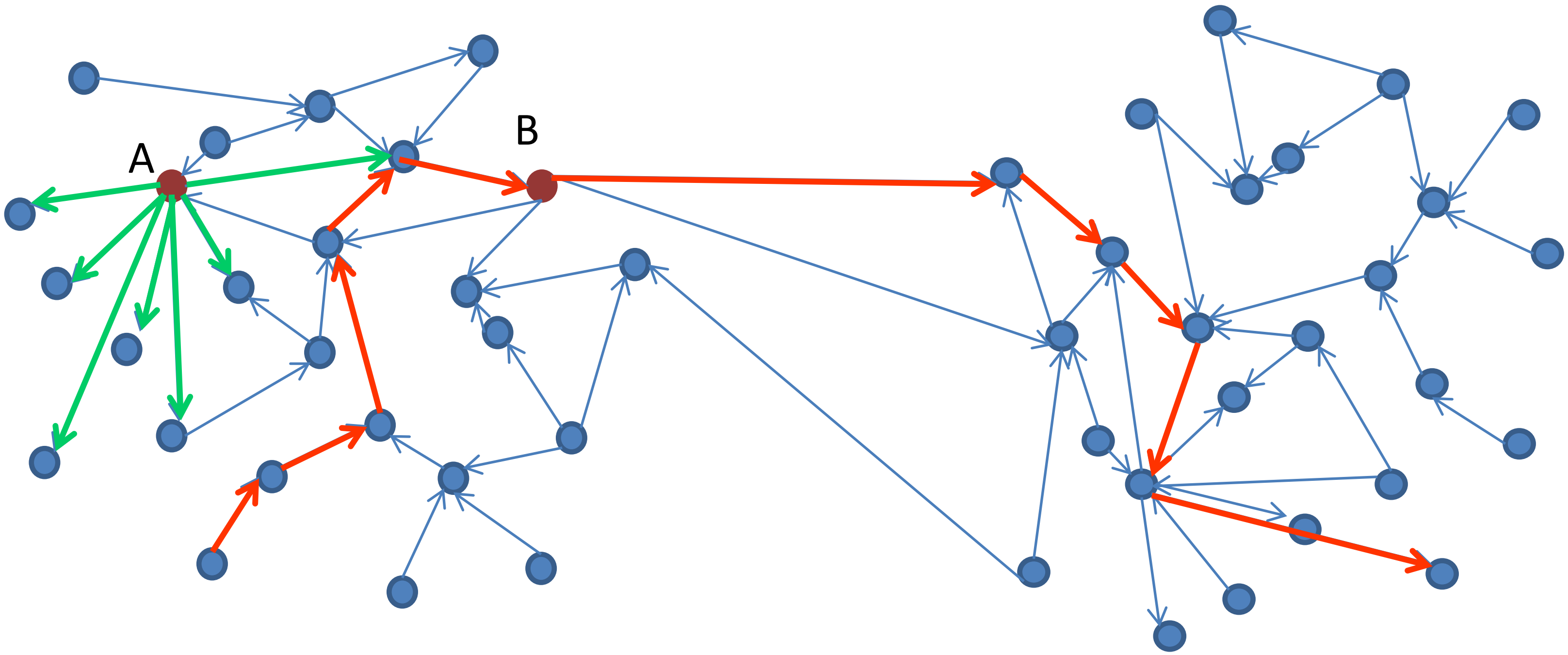
Degree Centrality v. Betweenness Centrality



- Is vertex A or B most central?
  - A has directed edges to more vertices (degree centrality)
  - B is on more shortest paths between vertex pairs (betweenness centrality)

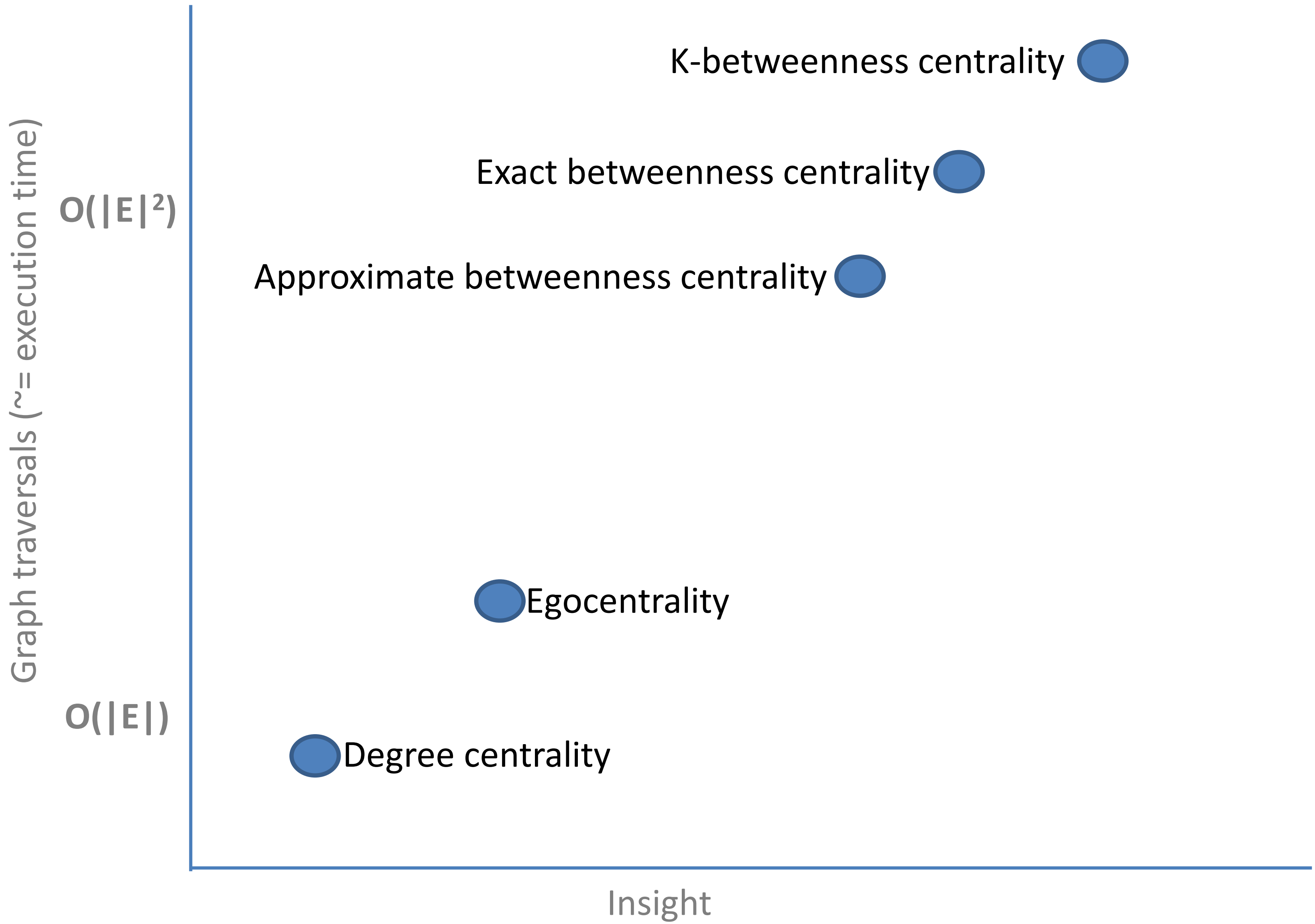
Local v. Global Metrics

Degree Centrality v. Betweenness Centrality

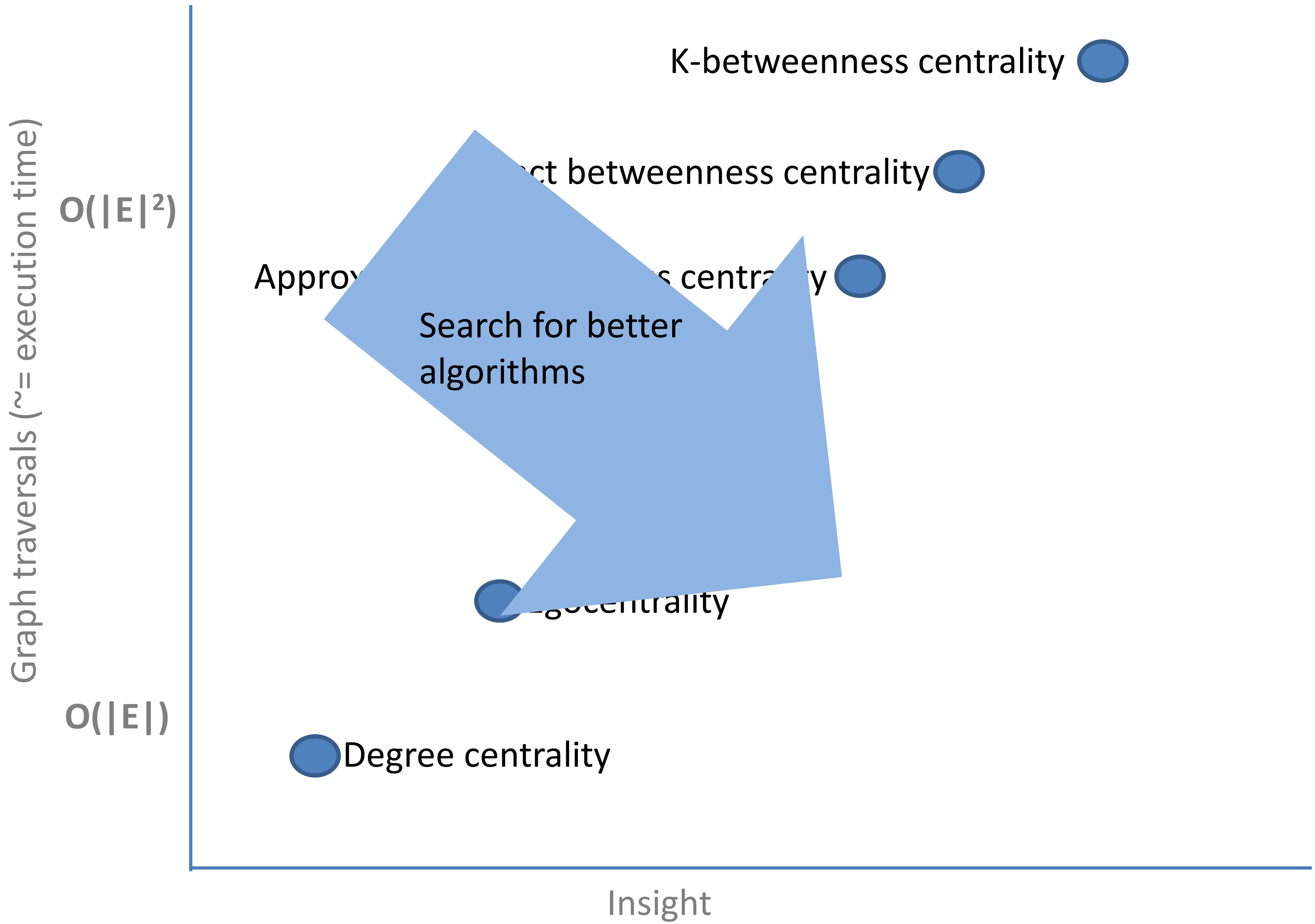


- Is vertex A or B most central?
  - A has directed edges to more vertices (degree centrality)
  - B is on more shortest paths between vertex pairs (betweenness centrality)

Algorithms: Insight v Graph Traversals



Algorithms: Insight v Graph Traversals



Knowledge Discovery Toolbox (KDT) Overview

- Target audiences
  - Primarily, (non-graph-expert) domain experts needing to analyze large graphs
  - Secondarily, graph-algorithm researchers and developers needing access to highly performant scalable graph infrastructure
- Target use cases
  - Broadly, problems needing the detail of algorithms that traverse the graph extensively
  - Social-network-based ranking and search
  - Homeland security
- Current KDT practicalities
  - Abstractions are (semantic) directed graph and sparse and dense vectors, all of which are distributed across a cluster
  - Python interface layered on Combinatorial BLAS
    - Delivers full scaling of CombBLAS with negligible Python overhead for non-semantic graphs
  - v0.2 release expected in October
    - x86-64 clusters running Windows or Linux
  - Open-source code available at `kdt.sourceforge.net` under New BSD license

Parsimony with New Concepts for Domain Experts

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering: Markov, and components
- Ranking: betweenness centrality, PageRank
- Matching: k-cycles

- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

Parsimony with New Concepts for Domain Experts

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering: Markov, and components
- Ranking: betweenness centrality, PageRank
- Matching: k-cycles

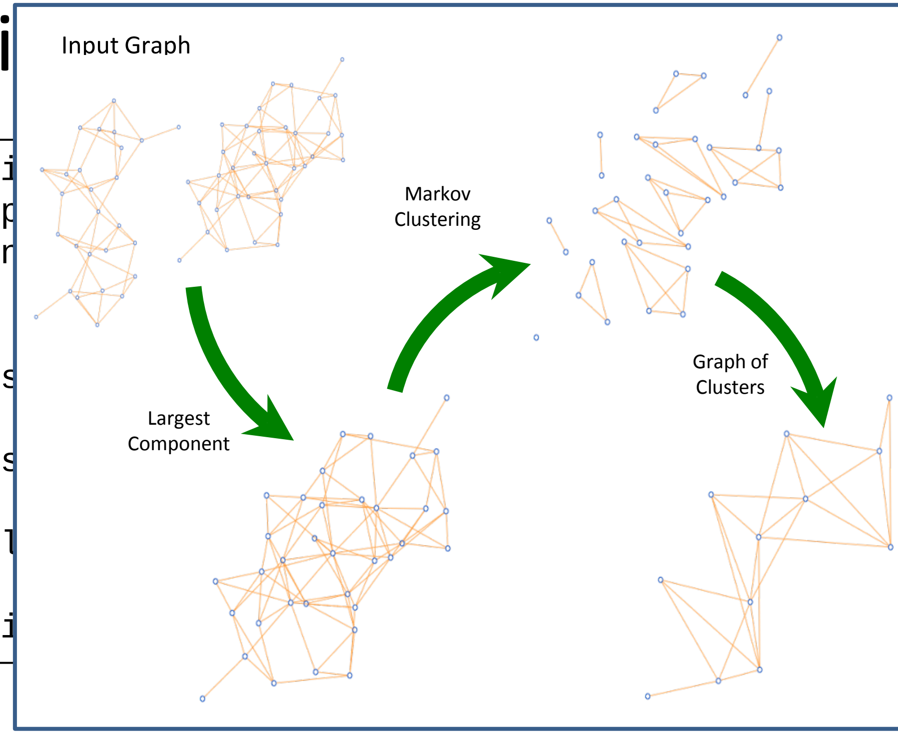
```
# bigG contains the input graph
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(comp==giantComp)

clus = G.cluster('Markov')

clusNedge = G.nedge(clus)

smallG = G.contract(clus)

# visualize
```

- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

Parsimony with New Concepts for Domai

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering: Markov, and components
- Ranking: betweenness centrality, PageRank
- Matching: k-cycles

```
# bi
comp
giar
G =

clus

clus

smal

# vi
```

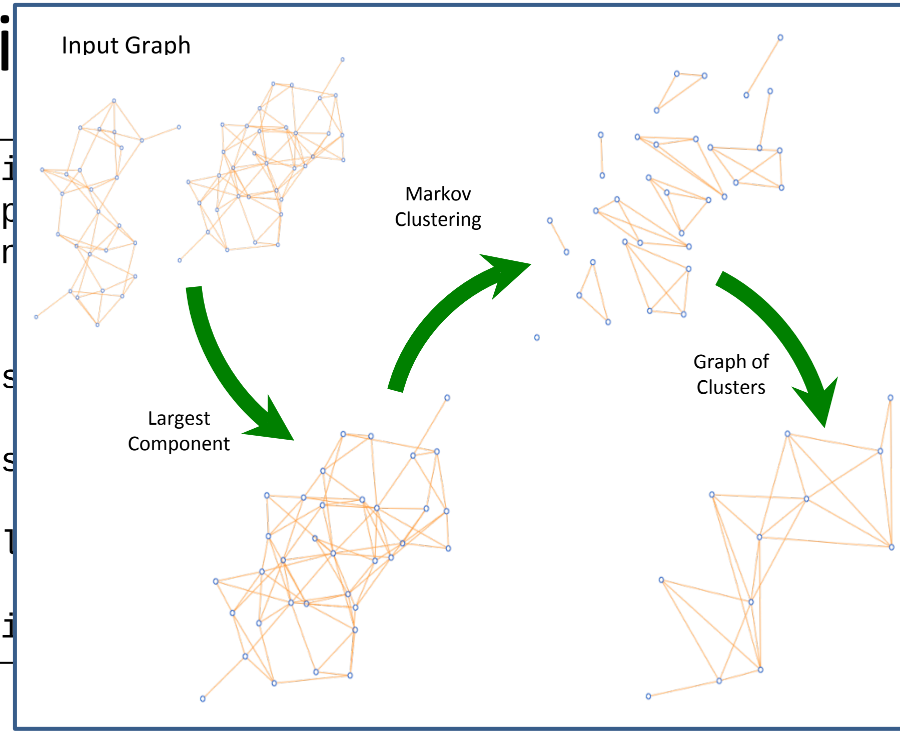


- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

Parsimony with New Concepts for Domai

- (Semantic) directed graphs
  - constructors, I/O
  - basic graph metrics (*e.g.*, `degree()`)
  - vectors
- Clustering: Markov, and components
- Ranking: betweenness centrality, PageRank
- Matching: k-cycles

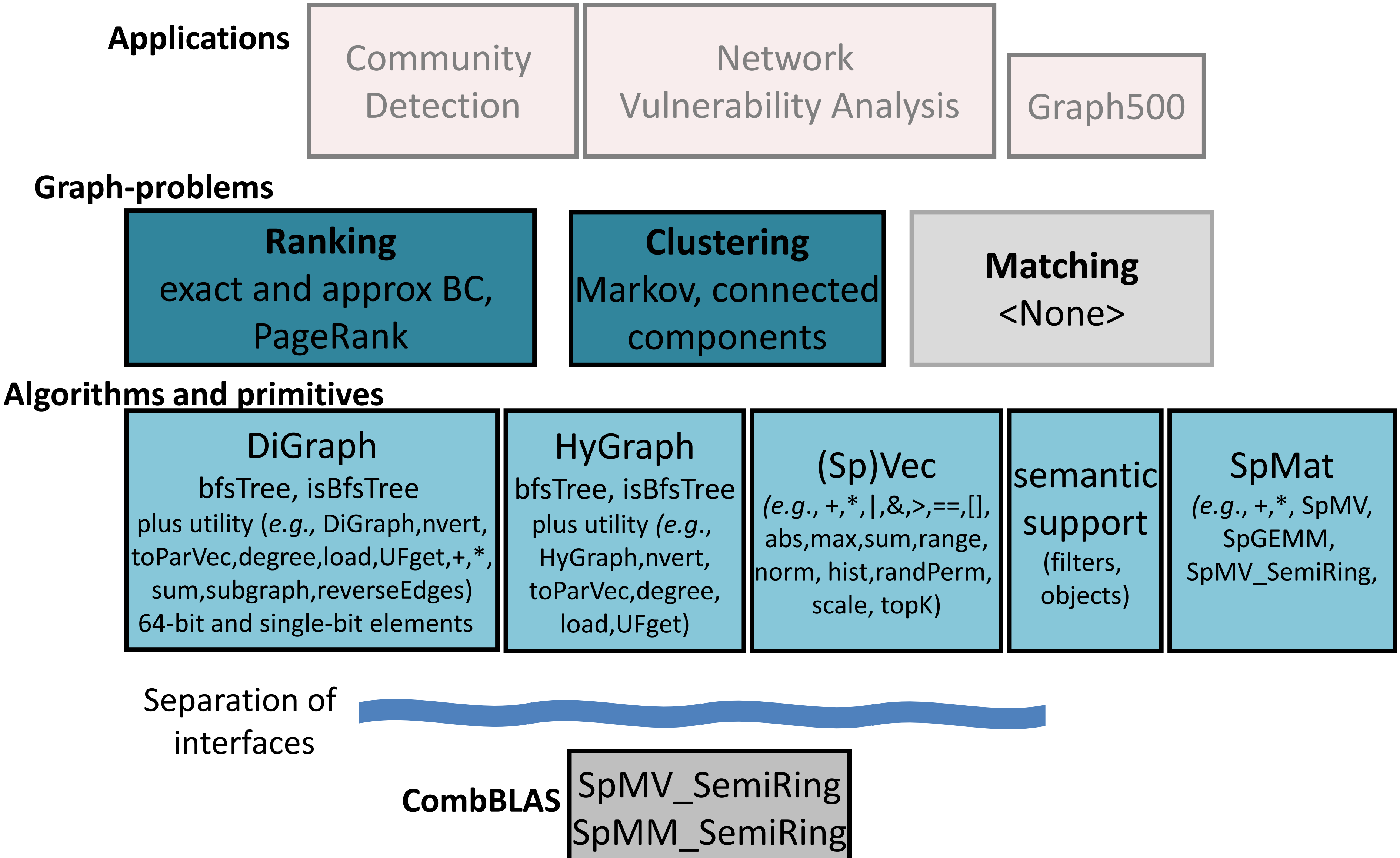
```
# bi
comp
giar
G =

clus

clus

smal

# vi
```



- Hypergraphs and sparse matrices
- Graph primitives (*e.g.*, `bfsTree()`)
- SpMV / SpGEMM on semirings

```
          [...]
L = G.toSpParMat()
d = L.sum(kdt.SpParMat.Column)
L = -L
L.setDiag(d)
M = kdt.SpParMat.eye(G.nvert()) – mu*L
pos = kdt.ParVec.rand(G.nvert())
for i in range(nsteps):
    pos = M.SpMV(pos)
```

Graph API (v0.2)



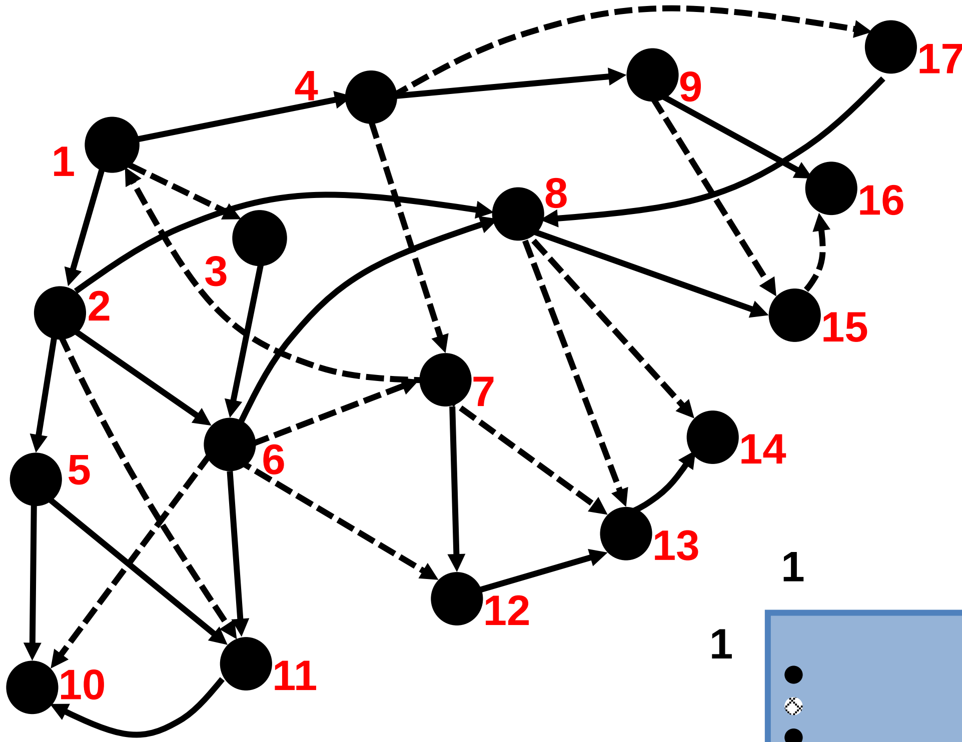
Semantic Graph Use Case

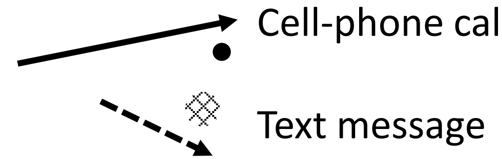
"Looking just at cell-phone communications, who are the leaders?"

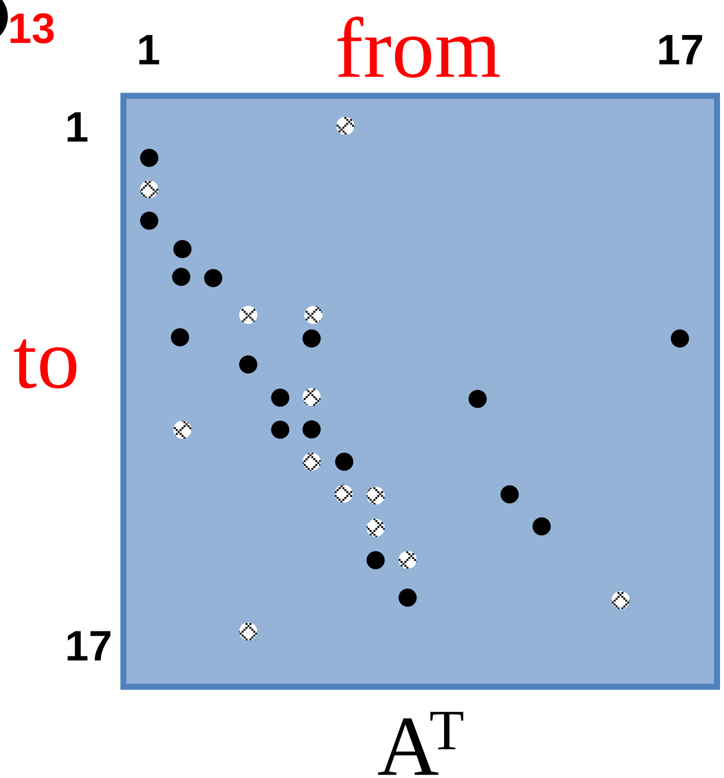
```
import kdt
# user function that converts a (file) record into an edge
def readRecord(self, sourceV, destV, record):
        sourceV = record[0]
        destV = record[1]
        self.category = record[2]
        self.type = record[3]
        return (sourceVert, destVert, self)
G = kdt.DiGraph.load('/file/my/graph/data', readRecord)

# edges for which the edge-filter returns True will
#   be used in the calculation
edgeFilter = lambda x:  x.category == CellPhone
G.addEFilter(edgeFilter)

# calculate leaders via approximate betweenness centrality
bc = G.centrality('approxBC')
leaders = bc.topK(10)
```

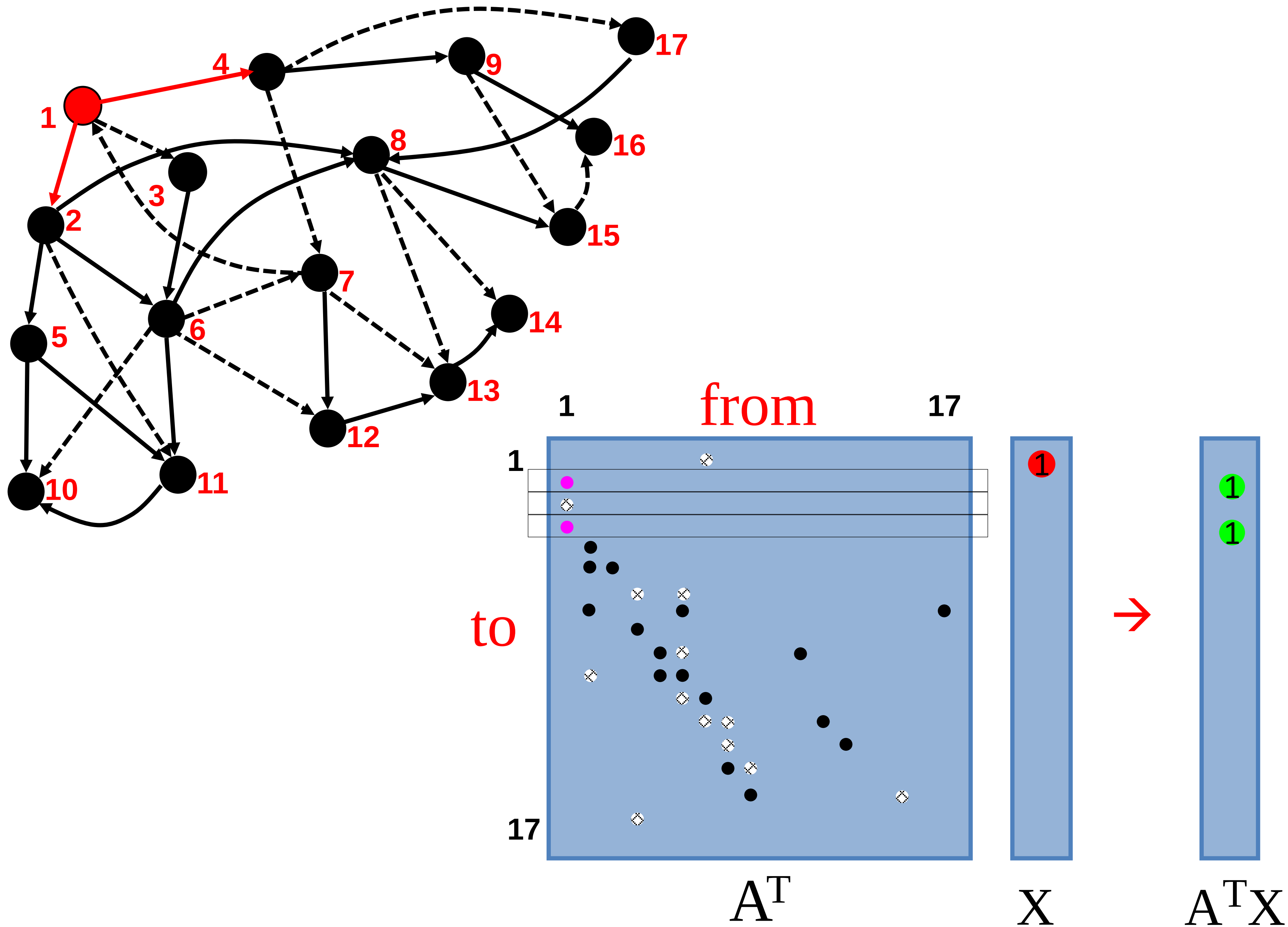
Caveat: Currently, expressing the filter in Python (rather than C++) leads to a big performance decrease; reducing/eliminating this decrease is work in progress.

Example Algorithm:
Find a breadth-first tree starting from a given vertex

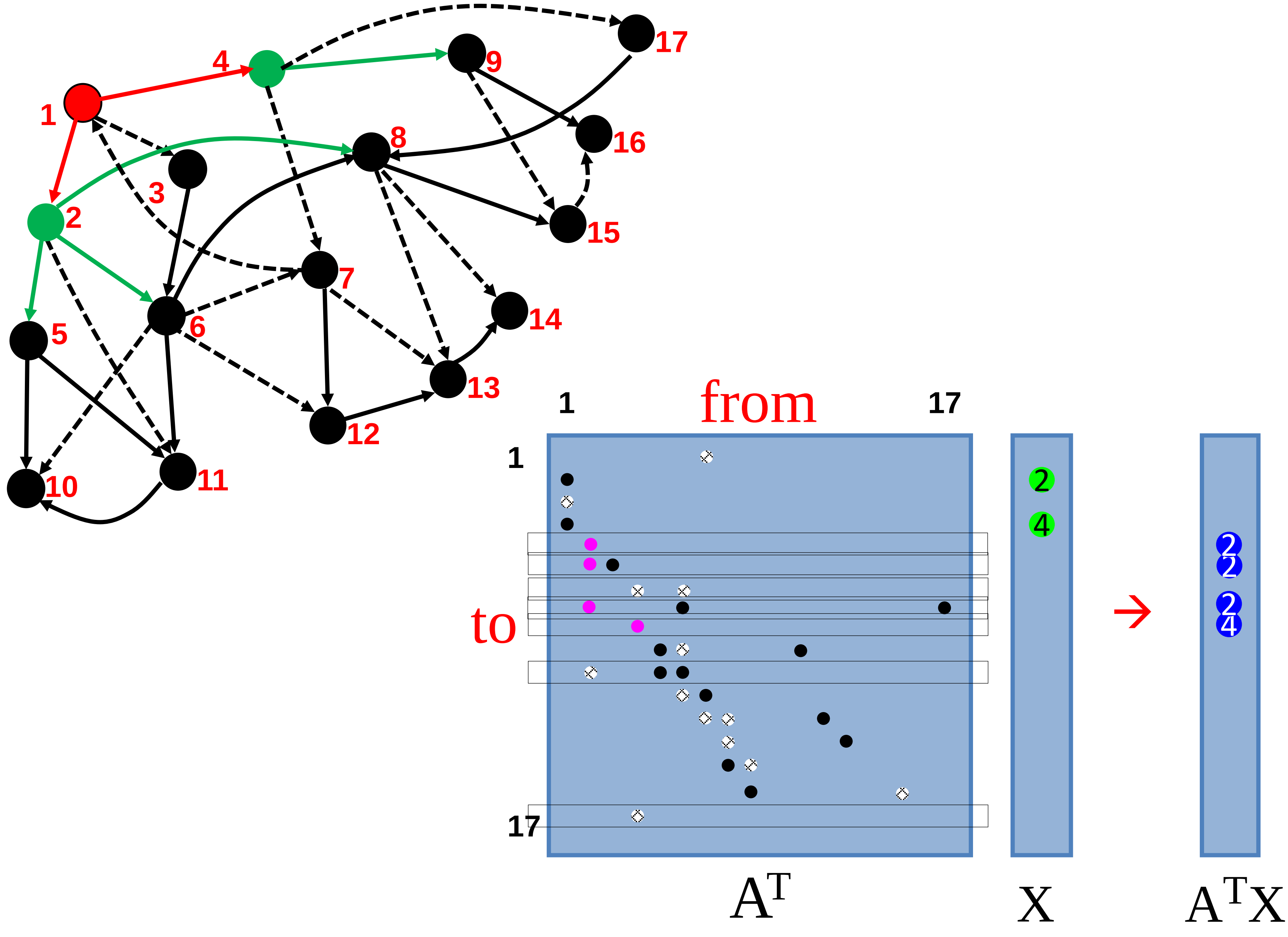


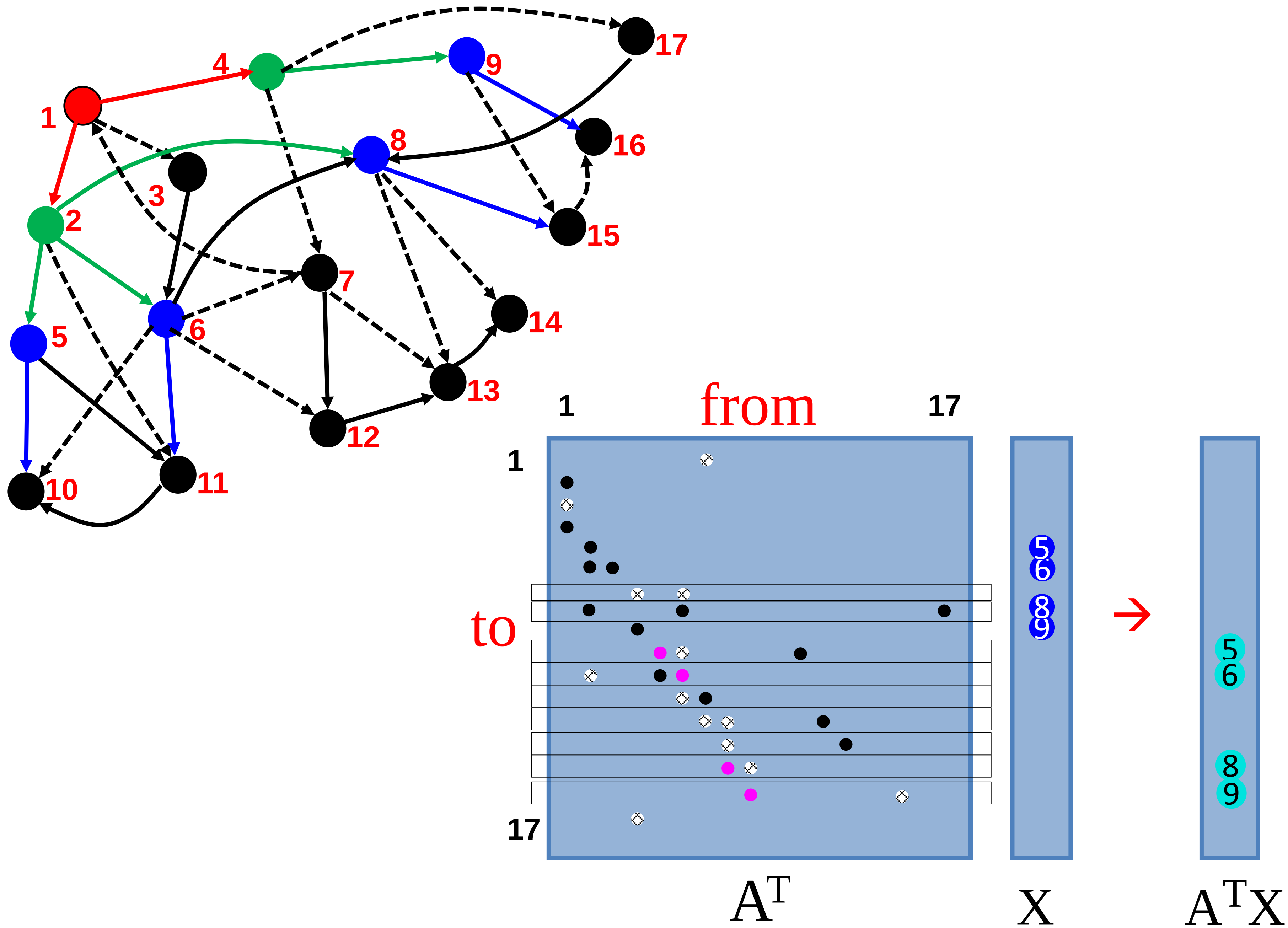




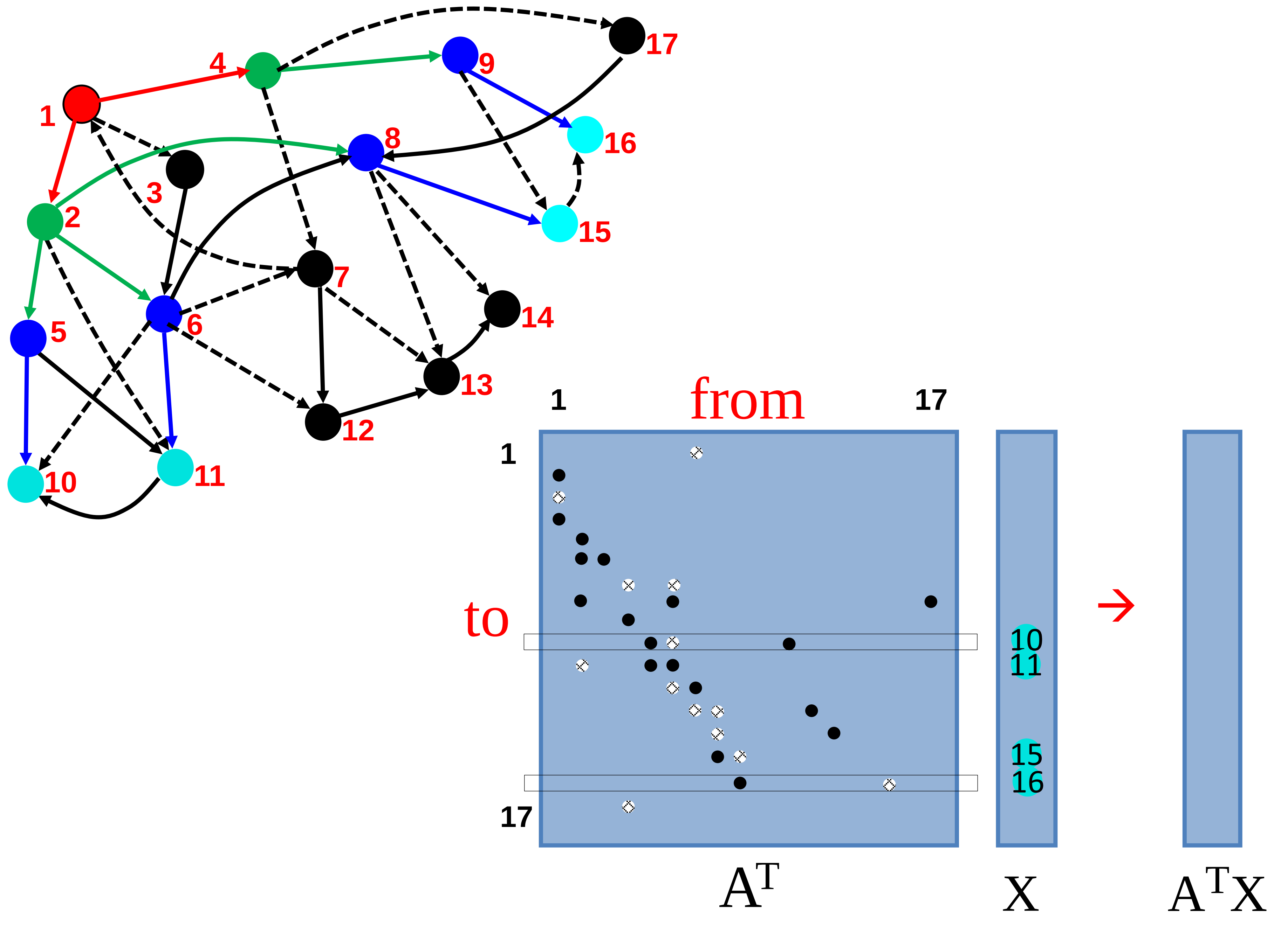


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
from
to
1
17
1
17
1
1
1
A^T
X
A^TX


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
from
to
1
17
1
17
2
4
2
2
2
4
A^T
X
A^T X



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
from
to
1
17
1
17
5
6
8
9
A^T
X
A^TX



1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
from
to
1
17
1
17
10
11
15
16
A^T
X
A^TX

The case for sparse matrices

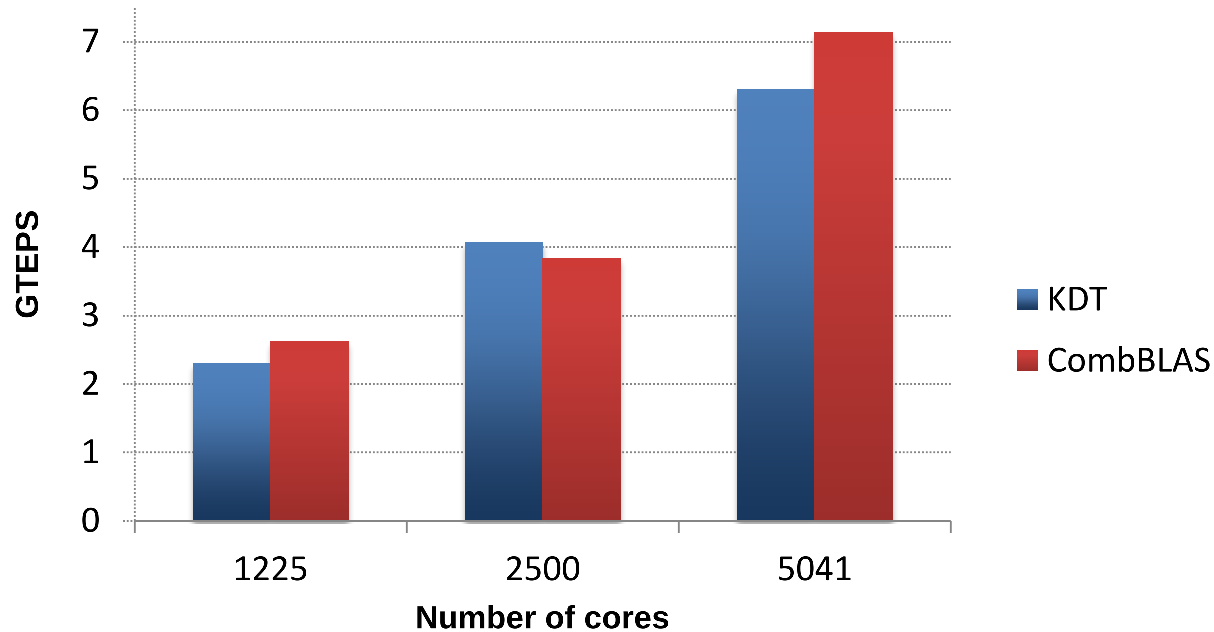
Many irregular applications contain coarse-grained parallelism that can be exploited by abstractions at the proper level.

Traditional graph computations	Graphs in the language of linear algebra
Data driven, unpredictable communication.	Fixed communication patterns
Irregular and unstructured, poor locality of reference	Operations on matrix blocks exploit memory hierarchy
Fine grained data accesses, dominated by latency	Coarse grained parallelism, bandwidth limited

UCSB

Performance

Graph500 in KDT or Combinatorial BLAS

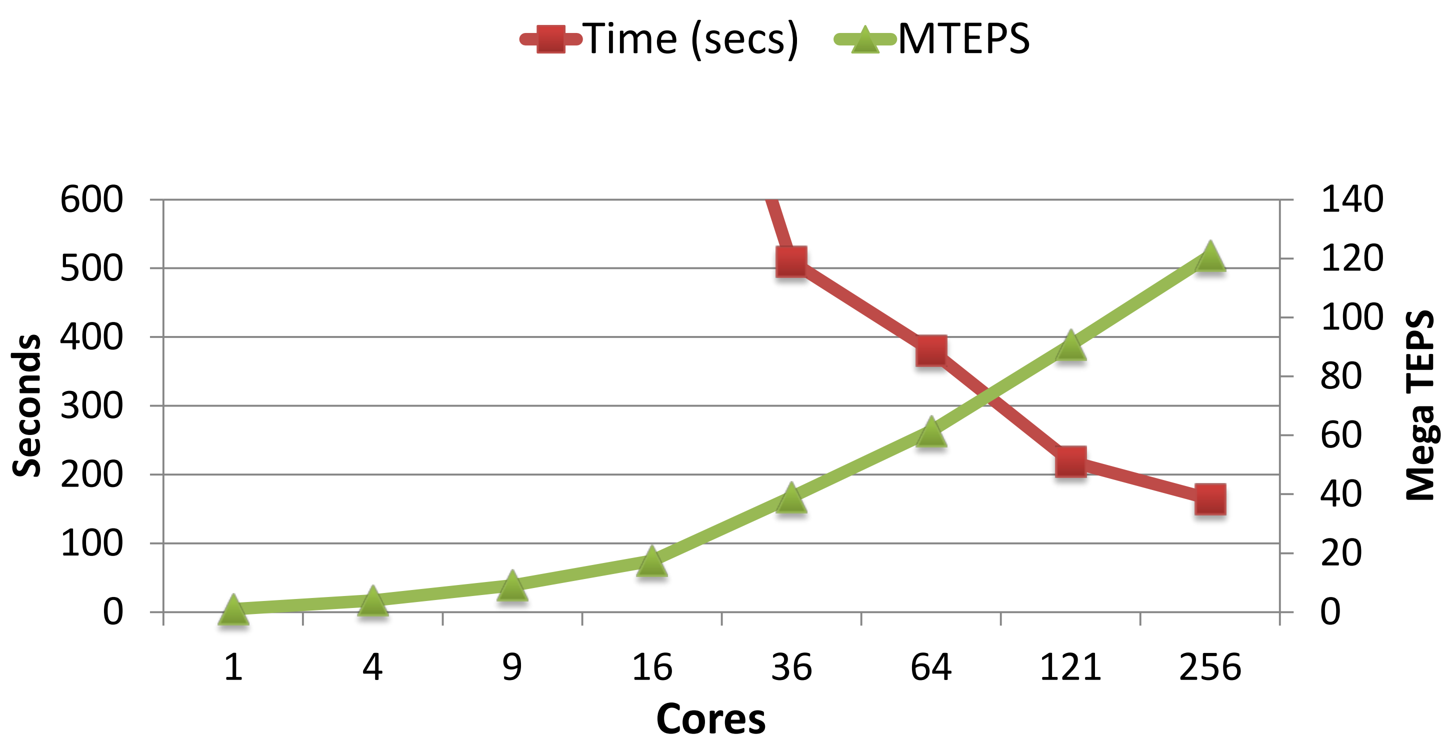


- Graph500 benchmark on 8B edges, C++ or KDT calling CombBLAS
- NERSC "Hopper" machine (Cray XE6)
- [Buluç & Madduri]: New hybrid of CombBLAS MPI + OpenMP gets 25 GTEPS on 2T edges (scale 37) on 43,200 cores of Hopper

Performance

Betweenness Centrality

- With a few hundred cores, can do even a complex graph analysis in near-interactive time
- 2M edges, approximate betweenness centrality sampling at 3%



Productivity

- Betweenness centrality
  – Python version initially written to SciPy interfaces
  – Porting to KDT took 11 hours for working, scalable implementation
- Markov clustering
  – Written by an undergraduate in 6 hours

Agenda

- Use cases and audience
- Technology
- Next steps

Next Steps

- Core technology
  - Evolve semantic graph support so fully usable
  - Implement support for streaming graphs
- Engineering
  - Couple with GUI / graph viz package
  - Port to Windows Azure
  - Accept more data formats
  - Extend coverage of clustering, ranking, and matching algorithms

KDT Summary

- Open-source toolbox targeted at domain experts
- Scalable to 10B-edge graphs and thousands of cores
- Limited set of methods, no graph viz yet
- kdt.sourceforge.net for details
- If you

  - have other use cases

  - need specific data formats or methods

  - have developed a method

  please contact me at steve.reinhardt@microsoft.com

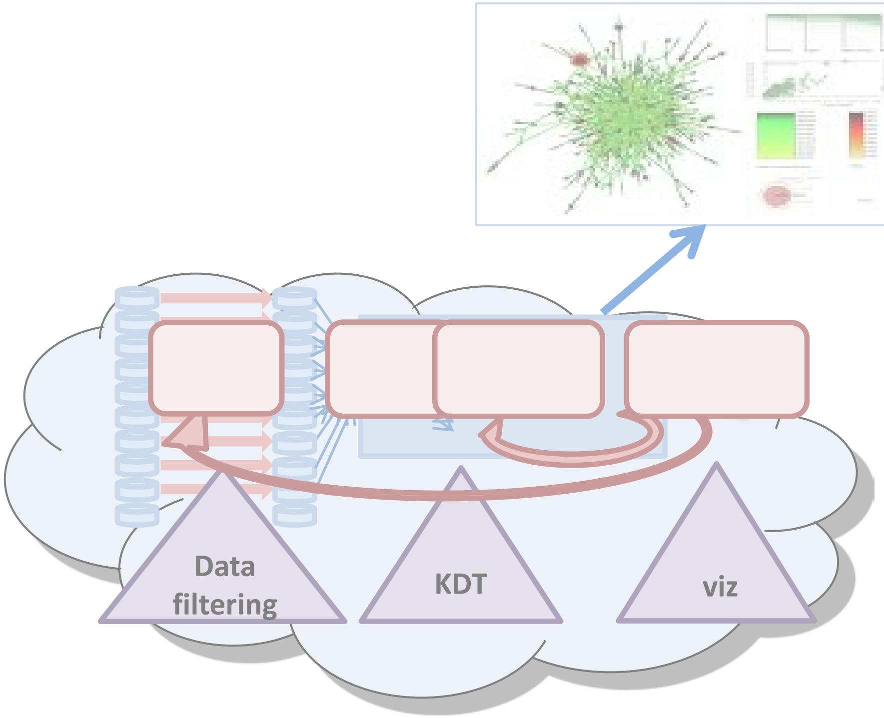
Knowledge Discovery Toolbox enables rapid algorithm development and fast execution for large-scale complex graph analytics

Backup

Further Info

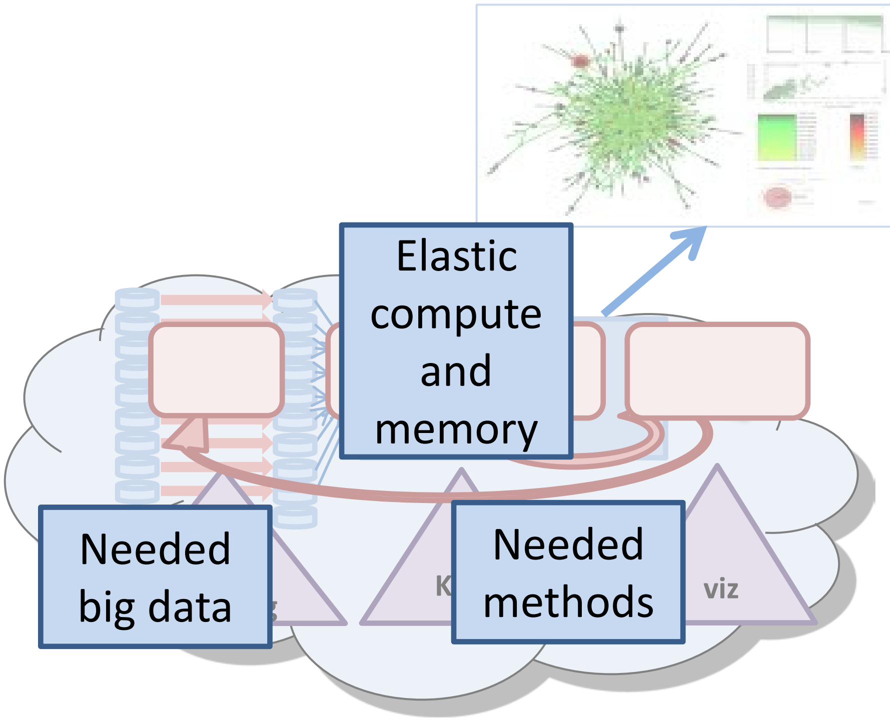
- Linked, by Albert-Laszlo Barabasi
- Graph Algorithms in the Language of Linear Algebra, by John Gilbert and Jeremy Kepner, SIAM

Cloud Benefits for Graph Analytics



Windows Azure

Cloud Benefits for Graph Analytics



- For domain expert
  - Elasticity of compute resource
  - Ready availability of needed data – what?
  - Ready availability of new methods – which?
- For graph-algorithm researcher
  - Quickly try your algorithm on big data
  - Quickly make it visible to domain experts

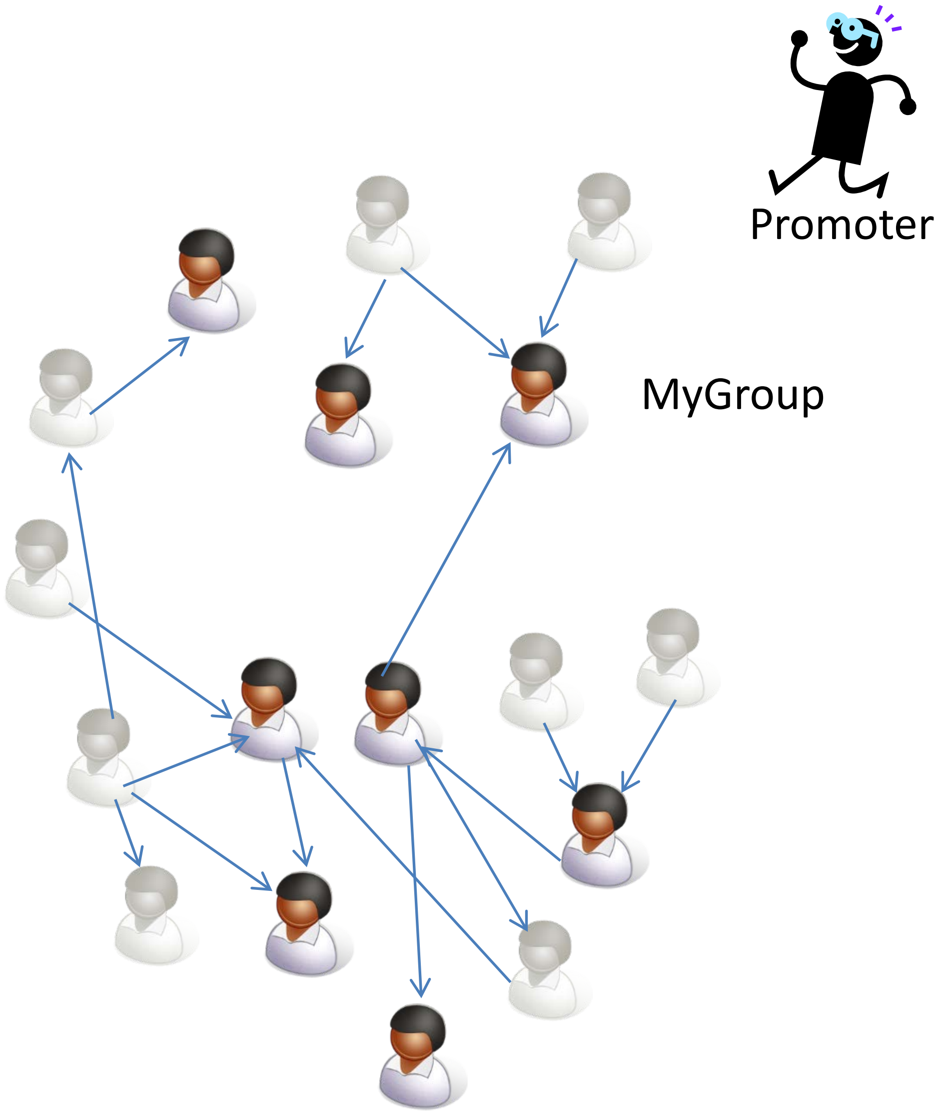
Windows Azure

"Transport of the mails, transport of the human voice, transport of flickering pictures -- in this century, as in others, our highest accomplishments still have the single aim of bringing men together."
Antoine de Saint-Exupery

Undelivered Possibilities

- Graph viz
- More ranking/clustering/matching options
- Availability in Azure
- Initial stages on disk, later stages in memory
- Dynamic/streaming graphs

Use Case: Find Influential People in a Social Network



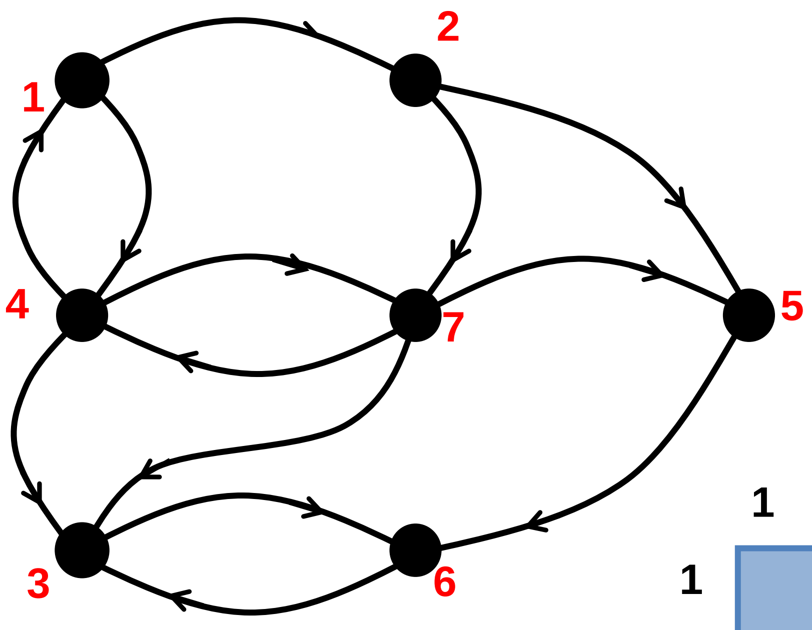
- Promoter has a SN group
- Wants to identify influencers on which to focus marketing efforts so as to maximize viral effect of the group
- Calls KDT with group name, gets back top N influencers
- Useful for (*e.g.*) viral marketing, public health

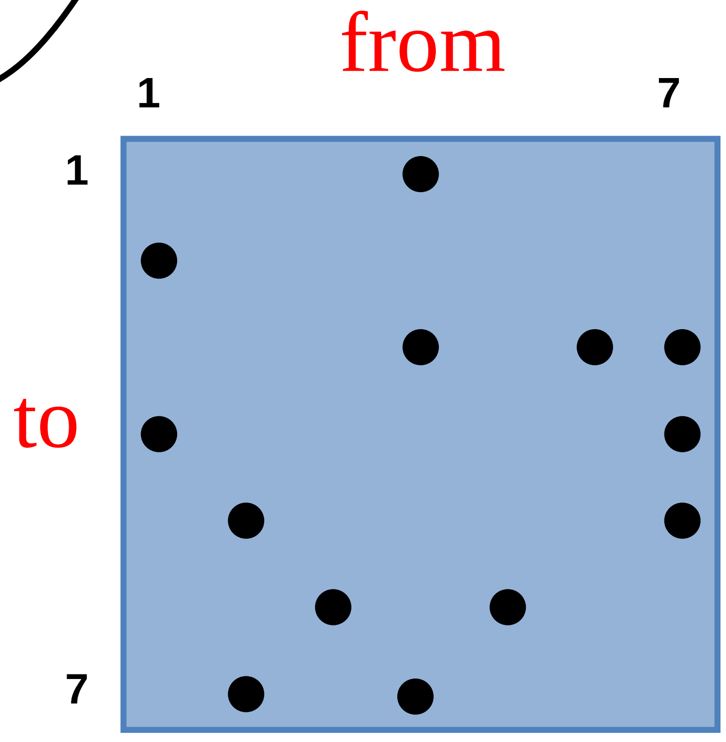
Comparison to Other Parallel Packages

Package	Target users		Interface	Supported memory*
	Graph-alg devs	Domain experts		
Pegasus	X		Hadoop	Distributed on-disk
Pregel	X		C++	Distributed on-disk
PBGL	X		C++	Distributed in-memory
MTGL	X		C++	Shared
SNAP (GA Tech)	X		C	Shared
SNAP (Stanford)	X	X	C++ / NodeXL	Shared
GraphLab	X		C++	Shared
CombBLAS	X		C++	Shared or distributed, in-memory
KDT	X	**X**	Python	Shared or **distributed**, in-memory

*"Shared" meaning either cache-coherent or Cray XMT-style

Example Implementation: bfsTree

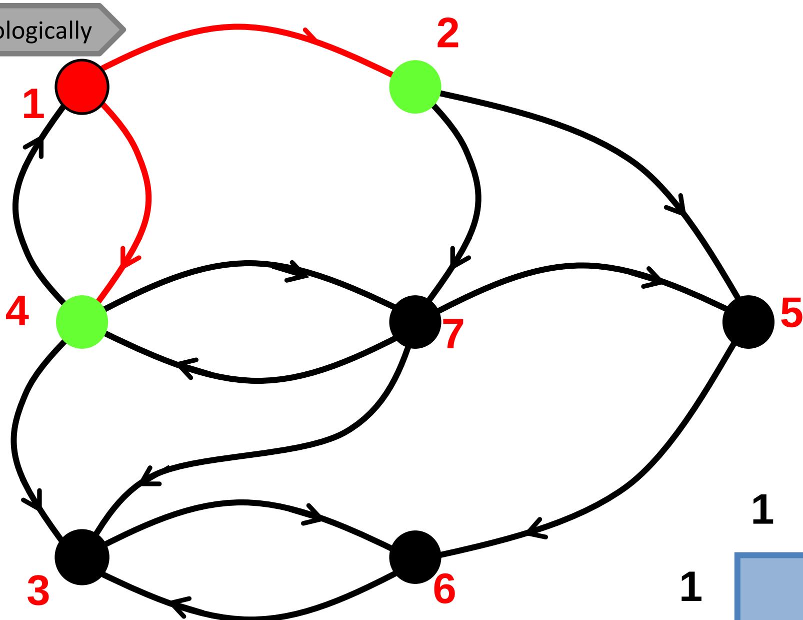


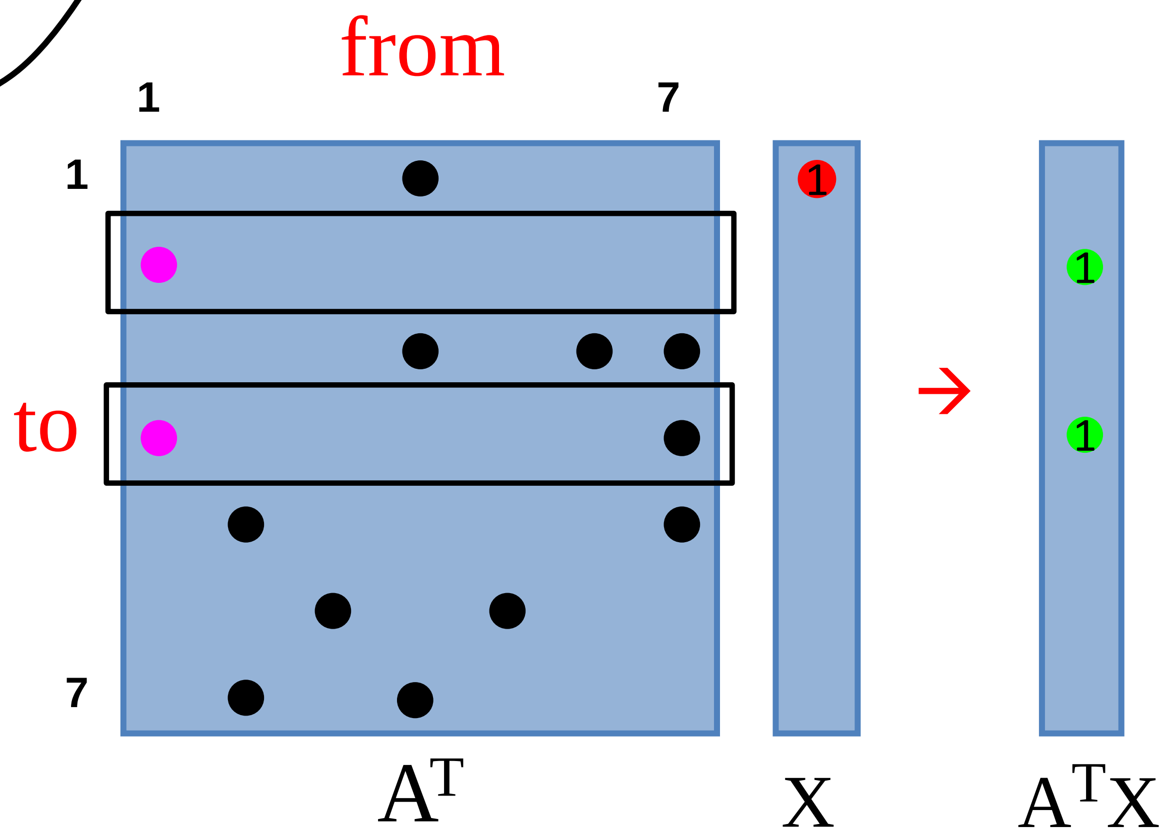


A^T

Technically

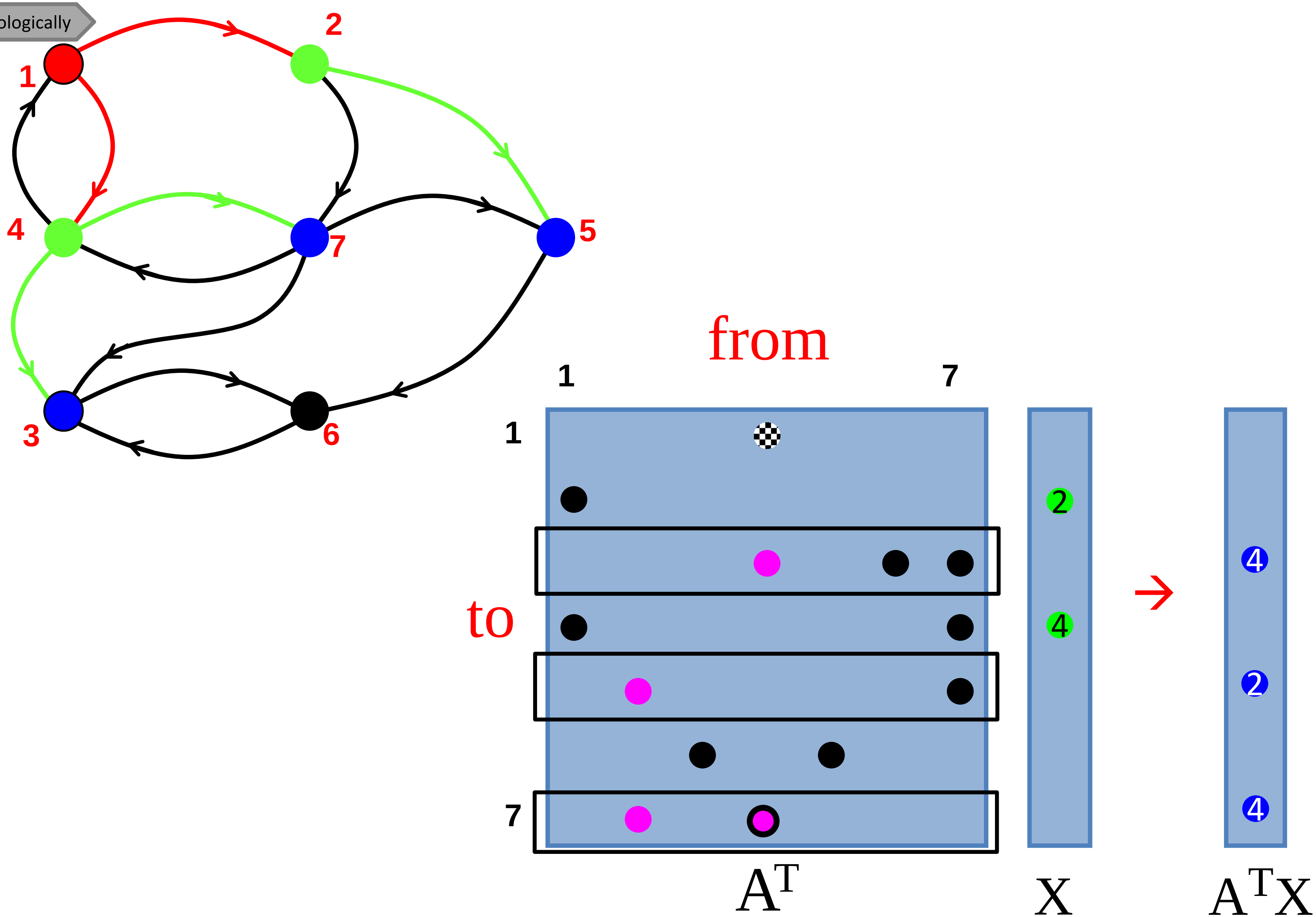
Ecologically

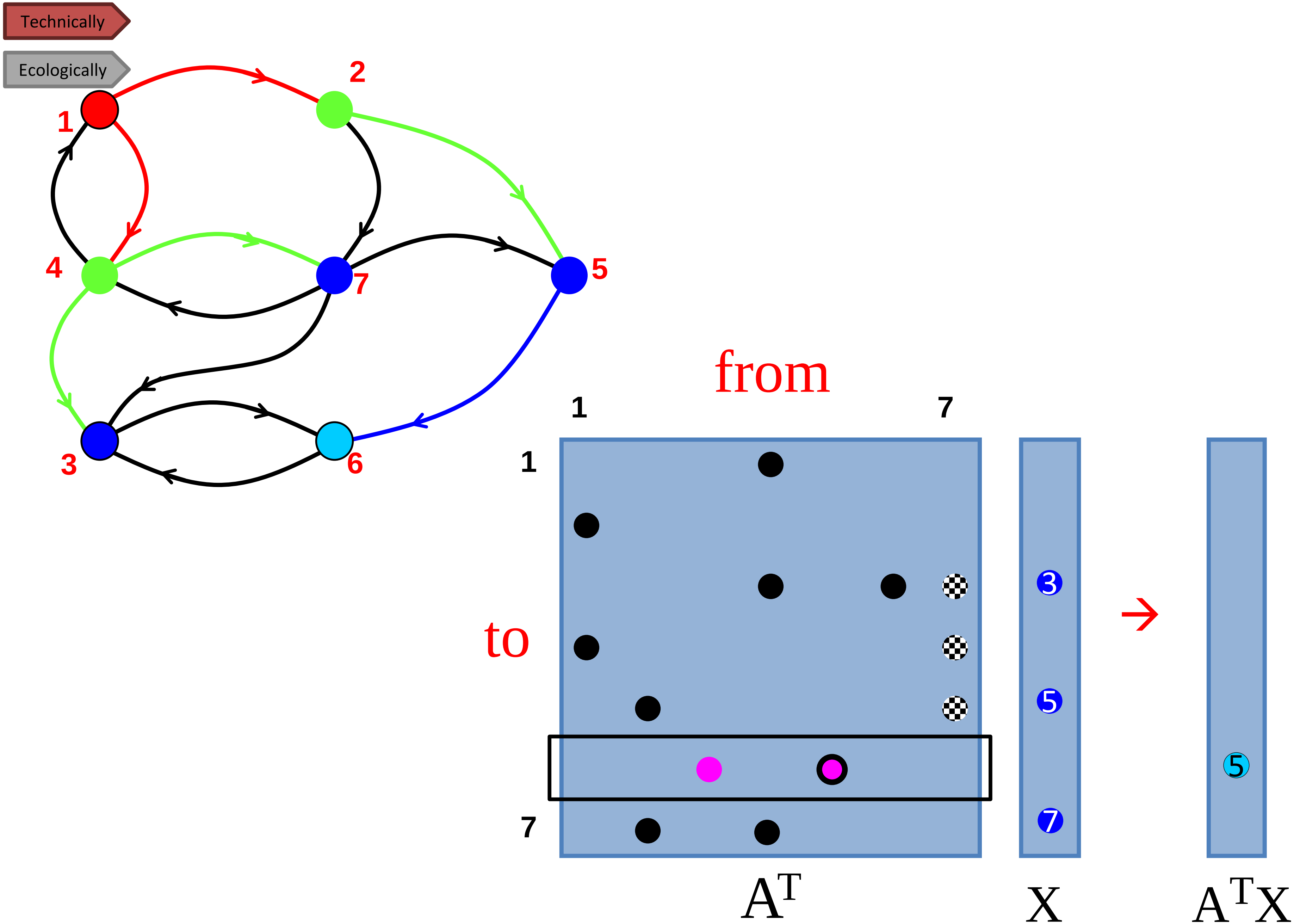




Technically

Ecologically

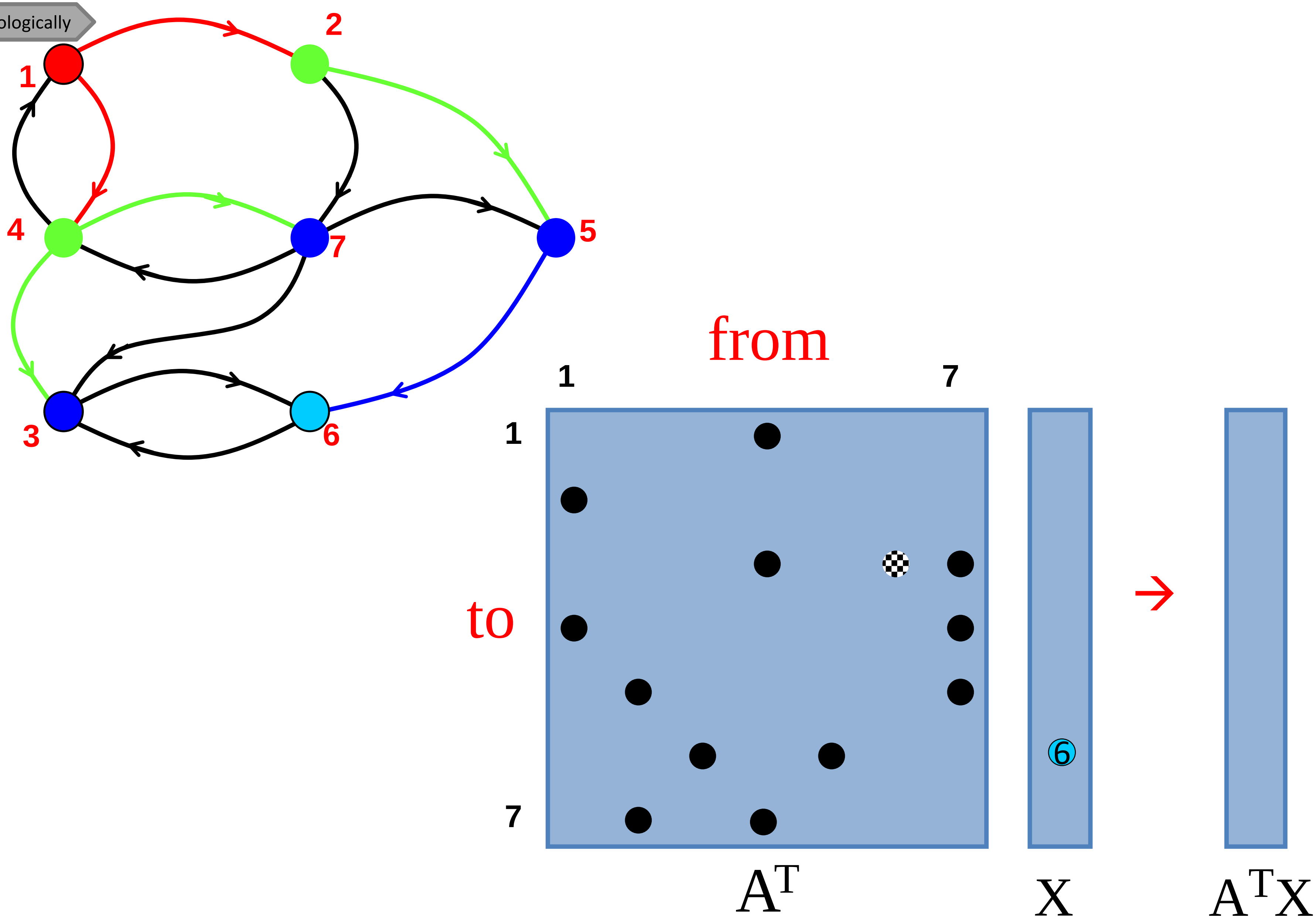




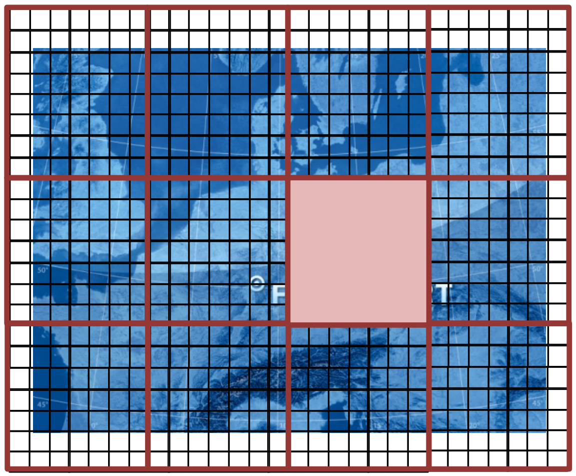
Technically
Ecologically
1
2
3
4
5
6
7
from
1
7
1
7
to
A^T
X
A^T X

Technically

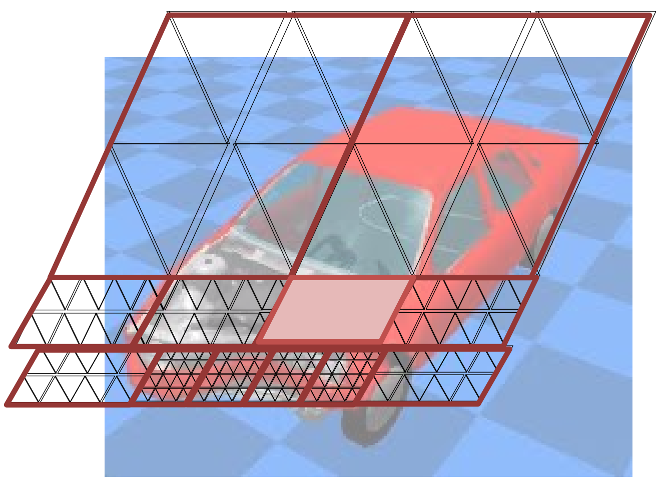
Ecologically



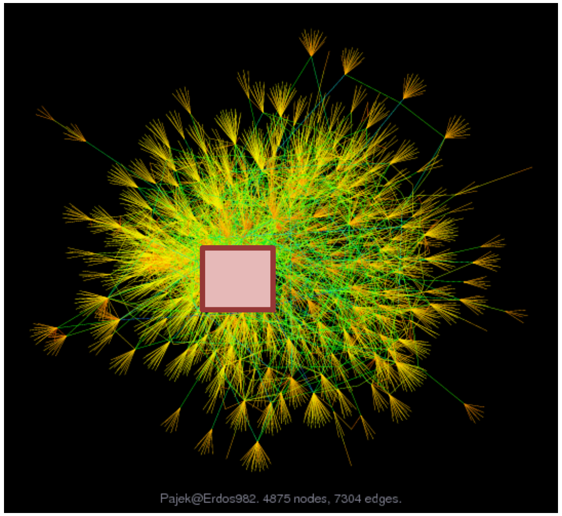
Many Graphs Don't Decompose Simply onto Distributed Memory



- 4n exchanges
- n^2 FLOPS
- Good locality



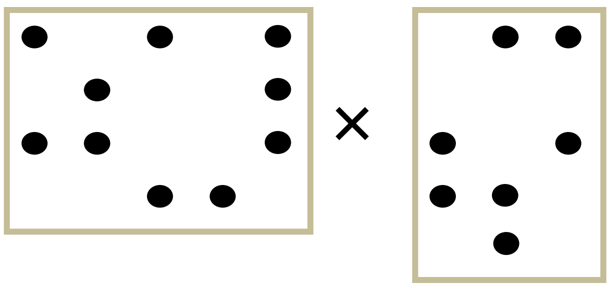
- 4n exchanges
- n^2 FLOPS
- Good locality



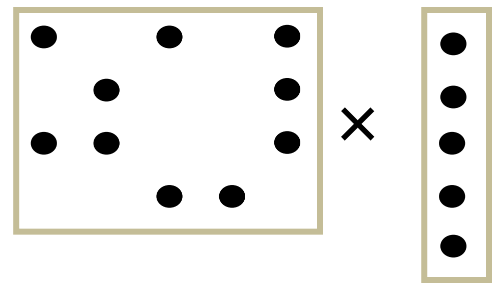
- ? exchanges
- ? OPS
- Usually poor locality, hence frequent comms, hence often a poor match for MapReduce

Sparse array-based primitives

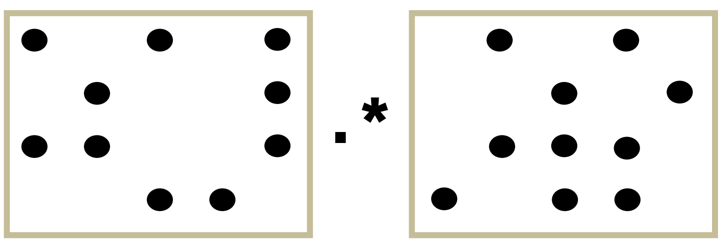
Sparse matrix-matrix multiplication (SpGEMM)



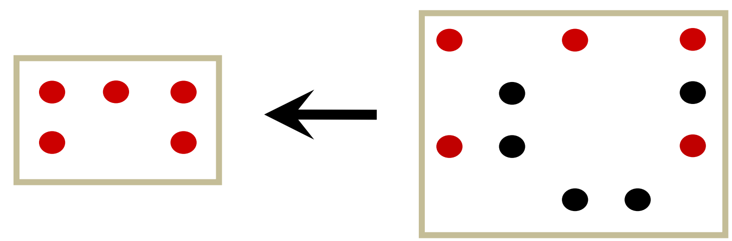
Sparse matrix-dense vector multiplication



Element-wise operations



Sparse matrix indexing



Matrices on various semirings: (x, +) , (and, or) , (+, min) , …

UCSB

Some Combinatorial BLAS functions

Function	Applies to	Parameters		Returns	Matlab Phrasing
SpGEMM	Sparse Matrix (as friend)	**A**, **B**: trA: trB:	sparse matrices transpose **A** if true transpose **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
SpMV	Sparse Matrix (as friend)	**A**: **x**: trA:	sparse matrices sparse or dense vector(s) transpose **A** if true	Sparse or Dense Vector(s)	$\mathbf{y} = \mathbf{A} * \mathbf{x}$
SpEWiseX	Sparse Matrices (as friend)	**A**, **B**: notA: notB:	sparse matrices negate **A** if true negate **B** if true	Sparse Matrix	$\mathbf{C} = \mathbf{A} * \mathbf{B}$
Reduce	Any Matrix (as method)	dim: *binop*:	dimension to reduce reduction operator	Dense Vector	sum(**A**)
SpRef	Sparse Matrix (as method)	**p**: **q**:	row indices vector column indices vector	Sparse Matrix	$\mathbf{B} = \mathbf{A}(\mathbf{p}, \mathbf{q})$
SpAsgn	Sparse Matrix (as method)	**p**: **q**: **B**:	row indices vector column indices vector matrix to assign	none	$\mathbf{A}(\mathbf{p}, \mathbf{q}) = \mathbf{B}$
Scale	Any Matrix (as method)	**rhs**:	any object (except a sparse matrix)	none	Check guiding principles 3 and 4
Scale	Any Vector (as method)	**rhs**:	any vector	none	none
Apply	Any Object (as method)	*unop*:	unary operator (applied to non-zeros)	None	none

bfsTree Implementation in KDT, for DiGraphs

(Kernel 2 of Graph500)

Technically

Ecologically

```
def bfsTree(self, root, sym=False):
    if not sym:
        self.T()     # synonym for reverseEdges
    parents = dg.ParVec(self.nvert(), -1)
    fringe = dg.SpParVec(self.nvert())
    parents[root] = root
    fringe[root] = root
    while fringe.nnn() > 0:
        fringe.spRange()
        self._spm.SpMV_SelMax_inplace(fringe._spv)
        pcb.EWiseMult_inplacefirst(fringe._spv,
            parents._dpv, True, -1)
        parents[fringe] = fringe
    if not sym:
        self.T()
    return parents
```

- SpMV and EWiseMult are CombBLAS ops that do not yet have good graph abstractions
  - pathsHop is an attempt for one flavor of SpMV

pageRank Implementation in KDT (p. 1 of 2)

Technically

Ecologically

```
def pageRank(self, epsilon = 0.1, dampingFactor = 0.85):
        # We don't want to modify the user's graph.
        G = self.copy()
        nvert = G.nvert()

        G._spm.removeSelfLoops()

        # Handle sink nodes (nodes with no outgoing edges) by
        # connecting them to all other nodes.
        degout = G.degree(gr.Out)
        nonSinkNodes = degout.findInds()
        nSinkNodes = nvert - len(nonSinkNodes)
        iInd = ParVec(nSinkNodes*(nvert))
        jInd = ParVec(nSinkNodes*(nvert))
        wInd = ParVec(nSinkNodes*(nvert), 1)
        sinkSuppInd = 0

        for ind in range(nvert):
            if degout[ind] == 0:
                # Connect to all nodes.
                for sInd in range(nvert):
                    iInd[sinkSuppInd] = sInd
                    jInd[sinkSuppInd] = ind
                    sinkSuppInd = sinkSuppInd + 1
        sinkMat = pcb.pySpParMat(nvert, nvert,
                       iInd._dpv, jInd._dpv,    wInd._dpv)
        sinkG = DiGraph()
        sinkG._spm = sinkMat
```

- This portion looks more like graph operations

pageRank Implementation in KDT (p. 2 of 2) (main loop)

Technically

Ecologically

```
G.normalizeEdgeWeights()
sinkG.normalizeEdgeWeights()

# PageRank loop
delta = 1
dv1 = ParVec(nvert, 1./nvert)
v1 = dv1.toSpParVec()
prevV = SpParVec(nvert)
dampingVec = SpParVec.ones(nvert) *
                 ((1 - dampingFactor)/nvert)
while delta > epsilon:
    prevV = v1.copy()
    v2 = G._spm.SpMV_PlusTimes(v1._spv) + \
             sinkG._spm.SpMV_PlusTimes(v1._spv)
    v1._spv = v2
    v1 = v1*dampingFactor + dampingVec
    delta = (v1 - prevV)._spv.Reduce(pcb.plus(),
                pcb.abs())
return v1
```

- This portion looks much more like matrix algebra

Graph500 Implementation in KDT (p. 1 of 2)

Technically

Ecologically

```
scale = 15
nstarts = 640

GRAPH500 = 1
if GRAPH500 == 1:
        G = dg.DiGraph()
        K1elapsed = G.genGraph500Edges(scale)

        if nstarts > G.nvert():
                nstarts = G.nvert()
        deg3verts = (G.degree() > 2).findInds()
        deg3verts.randPerm()
        starts = deg3verts[dg.ParVec.range(nstarts)]

G.toBool()

K2elapsed = 1e-12
K2edges = 0
for start in starts:
        start = int(start)
        if start==0:    #HACK:  avoid root==0 bugs for now
                continue
        before = time.time()
        parents = G.bfsTree(start, sym=True)
        K2elapsed += time.time() - before
        if not k2Validate(G, start, parents):
                print "Invalid BFS tree generated by bfsTree"
                print G, parents
                break
        [origI, origJ, ign] = G.toParVec()
        K2edges += len((parents[origI] != -1).find())
```

Graph500 Implementation in KDT (p. 2 of 2)

Technically

Ecologically

```
def k2Validate(G, start, parents):
        ret = True
        bfsRet = G.isBfsTree(start, parents)
        if type(ret) != tuple:
                if dg.master():
                        print "isBfsTree detected failure of Graph500 test %d" % abs(ret)
                return False
        (valid, levels) = bfsRet

        # Spec test #3:
        [origI, origJ, ign] = G.toParVec()
        li = levels[origI]
        lj = levels[origJ]
        if not ((abs(li-lj) <= 1) | ((li==-1) & (lj==-1))).all():
                if dg.master():
                    print "At least one graph edge has endpoints whose levels differ by
                                  more than one and is in the BFS tree"
                print li, lj
                ret = False

        # Spec test #4:
        neither_in = (li == -1) & (lj == -1)
        both_in = (li > -1) & (lj > -1)
        out2root = (li == -1) & (origJ == start)
        if not (neither_in | both_in | out2root).all():
                if dg.master():
                    print "The tree does not span the connected component exactly, root=%d" %
                              start
                ret = False

        # Spec test #5:
        respects = abs(li-lj) <= 1
        if not (neither_in | respects).all():
                if dg.master():
                    print "At least one vertex and its parent are not joined by an
                               original edge"
                ret = False

        return ret
```

- #1 and #2: implemented in isBfsTree

- #3: every input edge has vertices whose levels differ by no more than 1. Note: don't actually have input edges, will use the edges in the resulting graph as a proxy

- #4: the BFS tree spans a connected component's vertices (== all edges either have both endpoints in the tree or not in the tree, or source is not in tree and destination is the root)

- #5: a vertex and its parent are joined by an edge of the original graph