The Knowledge Discovery Toolbox (Version 0.1) User Guide

Last change date : February 15, 2011 **Revision : r0.06**

Open issues:

- Confirm start-up instructions (from shell) after SVN directories are moved around
- Examples for centrality() and cluster()
- PageRank interface

Contents

1 Overview

The Knowledge Discovery Toolbox (KDT) provides subject-matter experts with a simple interface to analyze very large graphs quickly and effectively without requiring knowledge of the underlying graph representation or algorithms. Domain experts are defined as experts in a field of study that is not graphs and graph algorithms, though they may have some familiarity with graph algorithms by using them. Because KDT is open-source, it can be customized or extended by interested (and intrepid) users.

Version 0.1

This early release provides a tiny selection of functions on directed graphs ranging from simple exploratory functions to complex algorithms. The current version works on graphs contained in the memory of multiple computers in a cluster (while hiding data representation and partitioning from the user). Notes specific to this version will be denoted by "***Version 0.1***" and appear in blue text.

1.1 Context

While graphs represent many real-world relationships in a mathematically robust way, their analysis with current methods does not scale. The modern "data tsunami" has created graphs in critical scientific and societal domains that are large enough to be prohibitively time-consuming to analyze with well-known methods. This has led graph-analysis experts to create more efficient graph analysis algorithms, but has also led to a gap between those experts and the non-graph subject-matter experts who need to use them. KDT counters the trend by exposing an API through the Python language that is efficiently and scalably implemented on computer clusters, while remaining suitable for domain experts by hiding the underlying implementation.

KDT is intended to accelerate a virtuous cycle among (a) subject-matter experts who need to analyze graphs that don't fit in the memory of a single computer node; (b) researchers working on improved graph algorithms; and (c) developers of tool infrastructure. We envision that subject-matter experts will do more analysis of large graphs with the current algorithms in KDT and provide feedback on which algorithms are (not) most useful for the large graphs in their domains. This will spur algorithm researchers and tool developers to develop new variants to analyze the subject-matter experts' graphs better.

We believe that many of the subject-matter experts don't know exactly what analysis they need to perform on their data, so they need to explore different algorithms and analyses. The KDT's goal is for the (intrepid) subject-matter expert (or her graduate student) to be able to compose building blocks at the Python level and explore interactively.

KDT's complex algorithms are difficult for even graph experts to verify, and so whenever possible KDT supports internal or user-driven verification of results. That variously consists of internal checks in KDT routines, companion routines that can validate a data structure (*i.e.*, `isBfsTree()` can validate the results of `bfsTree()`), and synthetic inputs whose metrics can be analytically derived (*i.e.*, each vertex of the graph created by `twoDTorus(nnodes)` has an identical betweenness centrality value of 0.5 * 2**(3*`scale`*0.5) – 2**`scale` + 1 for an even `nnodes`).

1.2 Intended use cases

Version 0.1

To tie KDT's development to the needs of potential users, this early release targets a small set of use cases. They all assume that the data about graph edges exists in "triplet" format, where the triplet is the source vertex, the destination vertex, and the attribute(s), such as weight and label, of the edge.

1.2.1 Creating a random power-law graph and calculating a breadth-first-search tree

Following the Graph500 benchmark, this use creates a random power-law graph in memory, calculates a breadth-first-search (BFS) tree from a starting vertex, and verifies the BFS tree.

1.2.2 Calculating the centrality of vertices of a graph

Centrality can be a step toward clustering that removes the most central vertices, but it is also an important metric in its own right for understanding which vertices most keep the graph connected. Exact betweenness centrality is simultaneously viewed by some researchers as not robust enough, because it provides information on only the single shortest path between two vertices, and by others as too computationally expensive, because its cost grows on the order of the number of vertices squared times the length of the longest path. For maximum flexibility, KDT provides a range of algorithms for calculating centrality from across the accuracy vs. execution-cost continuum.

1.2.3 Clustering the vertices of a graph

Clustering provides insight into which vertices are most associated by some criterion. As with centrality, KDT will later provide a few algorithms, believed to work best on very large data, that will be selected from many that have been proposed or implemented. Our current thinking is to implement Markov clustering first.

1.3 Deferred use cases

Version 0.1

KDT is envisioned to grow over time to support capabilities not included in this early release. The list of the deferred features includes: undirected graphs, multigraphs, and hypergraphs; general-purpose attributes, such as labels on vertices and edges; visualization of resulting graphs; support on other than x86-64 Linux clusters; and a disk-based implementation, for problems that do not fit in the memory of a computational cluster. The KDT developers believe that general-purpose attributes on edges and vertices are particularly critical to the applicability of the KDT, but we don't yet understand exactly what those attributes and operations on them should look like, so they are not in v0.1.

1.4 Legend and Naming Convention

In the function interface descriptions, required arguments are shown in black text, optional arguments in square brackets, and (optional) expert arguments are shown in grey.

```
cl = cluster('Markov',G[, nclus=k][, power=r])
```

For example, in the `cluster` function, `'Markov'` and `G` are required arguments, `nclus` is an optional argument, and `power` is an optional expert argument.

Names follow the Python convention of generally using lower case and capitalizing the first letter of a class name (`DiGraph`, e.g.), sometimes referred to as Pascal case. Multi-word member names follow the so-called camel case, where the first letter is lower case but the first letter of subsequent words is capitalized, such as `sendFeedback`.

1.5 Giving feedback to the KDT developers

KDT includes `sendFeedback` , a built-in feedback method that enables users to type in code that they wish KDT could execute and then send it to the developers. It uses IPython's `%logstart` facility to capture the code snippet. In Python code interpreters other than IPython the feedback mechanism will not work but will not otherwise obstruct program execution.

For sites that cannot directly send email onto the Internet, the default email address (in feedback.py, variable name `_kdt_Alias`) can be changed to an internal collection point.

The feedback mechanism can be used as follows:

In the course of solving your problem, when you need a function not implemented by KDT, type the code that you want KDT to support (which will evoke an error) and invoke its `sendFeedback` method. It will capture the most recent lines you've typed, create a file from them, and give you an opportunity to edit the file and add supporting comments. If you have feedback that is not directly related to a desired method, that can also be edited into the file. It will then prompt you for confirmation to send the file to the the KDT developers.

With a legend of user input / system responses / user annotation, an IPython session might look like:

```
In [43]: G = dg.Graph500Edges(scale=32)
In [44]: bc = G.centrality('approxBC',sample=0.01)
In [45]: # delete the top 10 most central vertices; topK and
In [46]: # deleteverts() methods don't exist
In [47]: discG = G.deleteverts(kdt.topK(bc,10))
In [48]: dg.sendFeedback()
The code example you want to send to the KDT developers is
in /home/sam/graphdev/KDT_email.
If you wish, edit it with your favorite editor. Type 'Send'
when you are ready to send it or 'Cancel' to cancel sending it.
>>>
```

[… If desired, edit the file with an editor …]

```
Send
In [49]:
```

2 Graph500 example

The Graph500 benchmark has replaced SSCA #2 [SSCA] as the primary benchmark for a segment of the graph-analysis research community. This new benchmark is “needed in order to guide the design of hardware architectures and software systems intended to support such applications and to help procurements. Graph algorithms are a core part of many analytics workloads.” The specification currently describes two kernels that, respectively, create the edge tuples and perform a breadth-first search of the graph from a start vertex. (Future kernels are expected to calculate single-source shortest path and the maximal independent set.) This section explicates the implementation of kernel 2 of the benchmark with KDT.

This section assumes that Python, IPython, and KDT have already been installed in their default locations. You can type the following to start a serial IPython session to run the Graph500 script and understand how it works. For instructions on how to run KDT and Graph500 in parallel with Python, see section 4.3.

```
[sam@neumann ~]$ which ipython
/usr/bin/ipython
[sam@neumann ~]$ ipython
Python 2.4.3 (#1, Nov 11 2010, 13:30:19)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints
more.

In [1]: import sys

In [2]: sys.path.append(‘/opt/kdt’)     [[FIX: double-check path]]

In [3]: import Graph500
Activating auto-logging. Current session state plus future input
saved.
Filename       : .KDT_log    #This output is from the startup of the IPython logstart
Mode           : over        # mechanism used for the sendFeedback function
Output logging : False       # from Section 1.5
Raw input log  : False       #      “”
Timestamping   : False       #      “”
State          : active      #      “”

Using fully connected graph generator
Graph500 benchmark run for scale =  8
Kernel 1 time =   0.0001 seconds
Kernel 2 time =   0.6300 seconds
                      0.0098 seconds for each of 64 starts
Kernel 2 TEPS = 6.6576e+06
```

```
In [4]: Graph500.parents

Out[4]: Elements stored on proc 0: {-1,196,215,215,215,215,215,215,
215,215,215,215,198,215,-1,215,185,249,215,22,113,-1,215,215,215,113,
215,215,147,215,31,185,215,215,184,122,31,215,224,83,233,-1,215,186,
                                [...]
113,215,215,-1,-1,185,215,163,}

In [5]:
```

Looking at the code in /opt/kdt/Graph500.py **[[FIX: double-check path]]**, we can see how the Graph500 specification is met with KDT methods.

```
import time
import numpy as np
import scipy as sc
import DiGraph as dg
#
#          [...]
#
#    previous kernels have created the following live variables
#    scale:  the log base 2 of the number of edges in the graph
#    edges:  the edges originally used to build the graph
#    G:      the directed graph
```

The following code selects vertices, each to be used as the root of a BFS tree, as specified by the benchmark. KDT-specific functions are shown in **blue**; overloaded Python operators are not highlighted.

```
# find the vertices of degree > 2
deg3verts = (G.degree(dir=dg.DiGraph.InOut) > 2).findInds()
nstarts = 64                    # #times to create a BFS
# randomly pick some of those vertices as start points
starts = np.random.randint(0,high=2**scale,size=(nstarts,))
k2Elapsed = 0
k2Edges = 0
```

The code segment below repeatedly calls the KDT `bfsTree` method, which creates a BFS tree for the graph and given starting vertex, and then validates the BFS tree via the user-written `k2Validate` function. Each element of the `parents` vector points to its (unique) parent in the tree.

```
for i in starts:
     start = deg3verts[i]
     before = time.clock()
     parents = G.bfsTree(start, sym=True)
     k2Elapsed += time.clock() – before
     if not k2Validate(G, start, parents):
          print “Invalid BFS tree generated by kdt.bfsTree”
     [origI, origJ, ign] = G.toParVec()
```

```
      K2edges += len((parents[origI] != -1).find())
```

The k2Validate user method calls the KDT isBfsTree method, which further illustrates the use of KDT with edge and vertex vectors. For instance, one section of isBfsTree code validates that each edge's endpoints are one level apart in the BFS tree. (Note that since isBfsTree is within the DiGraph class itself, which inherits the Graph class that contains the ParVec class, the code does not refer to the DiGraph module explicitly. Note further that this code is in /opt/kdt/DiGraph.py) **[[FIX: double-check path]]**.

```
# parents contains the source vertex for each tree edge, with the
#    exception of unreached vertices (parents[i]==-1) and
#    the root vertex (parents[i]==i)
treeEdges = (parents != -1) &
            (parents != ParVec.range(G.nvert()))
treeI = parents[treeEdges.findInds()]
# the sink vertex for each tree edge is the index, with the same
#    exceptions excluded as above
treeJ = ParVec.range(G.nvert())[treeEdges.findInds()]
if (levels[treeI]-levels[treeJ] != -1).any():
      ret = -2              # validation test #2 failed
      return
```

These very brief examples illustrate key points of KDT. First, the operations are graph operations, performed on graphs and (distributed edge- and vertex-) vectors. Second, to the extent practical, graph objects are accessible via standard Python methods such as subscripting, comparisons, and utility functions such as len.

3 Algorithms, Methods and Classes

Version 0.1

This early release of KDT supports only directed graphs with single edges between any two vertices, via the DiGraph class in the DiGraph module of the distributed memory version of KDT.

Collections of vertices and edges are represented as ParVec class instances. See section 3.3 for details about the class and method structure.

To save repetition, for the remainder of this section we assume that KDT has been imported as

```
      import DiGraph as dg
```

and that the DiGraph instance being operated on has the variable name G.

3.1 Algorithms

The KDT DiGraph class includes the algorithms in this section.

3.1.1 Centrality

Centrality is the degree to which, by some measure, a vertex is *central* to a graph. There is a wide variety of measures used and means of calculating those measures and hence numerous centrality algorithms.

Syntax

```
c = G.centrality('<algorithm>'[, <algorithm-specific keyword arguments>])
```

Description

The `centrality` function takes as input a `DiGraph` object and an algorithm identifier (see below) and returns a `ParVec` (of length equal to the number of vertices in the graph) that contains, for each respective vertex of the graph, the vertex's centrality value. Optional algorithm-specific keyword arguments may also be specified as described by the algorithm-specific sections below.

3.1.1.1 Exact betweenness centrality algorithm

Betweenness centrality is the degree to which a vertex is *between* all other vertices in the graph, calculated as the fraction of shortest paths between two vertices that pass through the given vertex [Freeman 1977].

Syntax

```
cl = G.centrality('exactBC'[, normalize=True])
```

Description

The `exactBC` algorithm calculates exact betweenness centrality on the graph. The optional algorithm-specific `normalize` argument, which defaults to True, causes the function to normalize the betweenness values by dividing each value by (#vertices-1)*(#vertices-2).

Performance

Exact betweenness centrality has computational complexity of O(#vertices**2 * diameter(graph)), so it can be prohibitively expensive for large graphs, while approximate betweenness centrality is less computationally expensive.

3.1.1.2 Approximate betweenness centrality algorithm

This algorithm [Bader] approximates betweenness centrality for each vertex by using a sample of vertices between which to calculate the fraction of shortest paths, rather than using all vertices as in exact betweenness centrality.

Syntax

```
cl = G.centrality('approxBC'[, normalize=True, sample=])
```

Description

The `approxBC` algorithm performs approximate betweenness centrality on the graph. The optional algorithm-specific `normalize` argument, which defaults to True, causes the function to normalize the betweenness values by scaling each value by the inverse of the `sample` factor (# total vertices / (#vertices actually calculated) and the inverse of (#vertices-1)*(#vertices-2). The optional `sample` expert argument is a floating-point value between 0 and 1 that denotes the fraction of vertices whose connecting paths are calculated in the approximation; a value of 1.0 equates to exact betweenness centrality; the default value is 0.05.

Performance

Approximate betweenness centrality has computational complexity of O(sample * #vertices**2 * diameter(graph)).

3.1.2 Single-membership Clustering [FIX: clustering out for v0.1?]

Clustering discovers the similarity of groups of vertices in the graph and assigns each vertex to a cluster of similar vertices. Numerous algorithms have been proposed, most of whose efficacy on large graphs has not been well explored, and so users may want to try different algorithms to find the one that works best for a particular dataset. One form of clustering restricts any vertex to being a member of only a single cluster and is implemented via the `cluster` function.

Syntax

```
cl = G.cluster('<algorithm>'[, <algorithm-specific keyword arguments>])
```

Description

The `cluster` function takes as input a `DiGraph` object and an algorithm (see below) and returns a `ParVec` (of length equal to the number of vertices in the graph) that contains, for each respective vertex of the graph, which of the discovered clusters it belongs to. The clusters are numbered from 0 to the number of clusters-1. Optional algorithm-specific keyword arguments may also be specified as described by the algorithm-specific sections below.

3.1.2.1 Markov clustering algorithm

Markov clustering [vanDongen, link], also known as random-walk clustering, works by repeatedly expanding the flow of a network and inflating local connections, through which the clusters emerge.

Syntax

```
cl = G.cluster('Markov'[, power=r])
```

Description

The `Markov` algorithm performs Markov clustering on the graph. There is no need to specify the number of clusters, as that value emerges from the algorithm. The expert `power` argument controls the rate of convergence of the algorithm; higher values lead to faster convergence by favoring smaller clusters over larger clusters.

3.1.2.2 Modularity clustering algorithm

Modularity clustering or “community detection” [Girvan] determines clusters by calculating which edges are least central to clusters and hence most between clusters; removing those edges exposes the clusters.

Syntax

```
cl = G.cluster(‘modularity’)
```

Description

The `modularity` algorithm performs modularity clustering on the graph.

3.1.3 Search

Several graph algorithms search for connectedness within the graph, such as trees, connected components, independent sets, paths, etc.

3.1.3.1 Breadth-first Search Tree

The `bfsTree` function creates a breadth-first search tree of a `DiGraph` instance from a starting vertex.

Syntax

```
parents = G.bfsTree(start[, sym=False])
```

Description

The BFS function takes as input a `DiGraph` instance and the index of the vertex from which to start the search. It returns a `ParVec` (of length equal to the number of vertices in the graph) that denotes, for each respective vertex of the graph, the vertex’s parent in the BFS tree. The `start` vertex’s parent is itself. A vertex that is unreachable from the `start` vertex has a parent of -1. The optional expert argument `sym` denotes whether the graph is a symmetric graph, which if True enables the operation to run faster.

3.2 General-purpose methods

The KDT `DiGraph`, `ParVec`, and `SpParVec` classes include the general-purpose methods in this section.

3.2.1 Built-in methods for ParVec vectors

Version 0.1

The `ParVec` class represents vertex and directed-edge vectors. Its implementation for distributed memory includes a significant but not complete set of methods, which are mapped onto standard Python operators or methods consistent with clear understandability.

Note: In version 0.1, `ParVec` instances only support a 64-bit floating-point or integer elemental datatype, and no explicit Boolean datatype. Any `ParVec` whose elements are each either 0 or 1, such as “`dg.ParVec.range(n) < k`”, will be viewed as a Boolean vector for indexing purposes. The isBool() method tests this characteristic.

Syntax

```
# v, v2, and v3 are ParVec instances
# k is an integer scalar, m is an integer or floating-point scalar
v2 = v > k              # v2 is a Boolean vector, where each element
v2 = v >= k             #  is the comparison of the corresponding
v2 = v < k              #  element of v with the scalar k
v2 = v <= k
v2 = v == k
v2 = v != k
v2 = v > v3             # v2 is a Boolean vector, where each element
v2 = v >= v3            #  is the comparison of the corresponding
v2 = v < v3             #  elements of v and v3
v2 = v <= v3
v2 = v == v3
v2 = v != v3
v2 = ~v                 # negation (only for Boolean vectors)
v3 = v & v2
v3 = v | v2

v3 = v + k
v3 = v + v2
v += k
v += v2
v3 = -v
v3 = v – k
v3 = v – v2
v -= k
v -= v2
v3 = v * k
v3 = v * v2
v3 = v / k
v3 = v / v2
v3 = v % k
v3 = v % v2

v[k] = m
m = v[k]
v[vec] = m              # vec an integer vector
v2 = v[vec]

v.abs()
v2 = abs(v)
```

Description

These methods have their standard definitions in Python with the following exceptions:

- Only a key set of indexing (“[]”) modes are supported. [[**FIX**: update]]
    - Indexing on the right-hand side of an equation by a scalar
    - Indexing on the right-hand side of an equation by a non-Boolean `ParVec` instance
    - Indexing on the left-hand side by a scalar with a scalar value
    - Indexing on the left-hand side by a Boolean `ParVec` index vector with a value the same length as the `ParVec` instance being modified. Only the value elements corresponding to the True elements in the index vector are referenced.

3.2.2 Non-built-in methods for ParVec vectors

The `ParVec` class implements several other methods, some of which are common Python names and some which are not.

Syntax

```
# v and v2 are ParVec instances, m a scalar
v2 = v.copy()              # deep copy
vSparse = v.toSpParVec()   # convert to a (sparse) SpParVec

v2 = v.ceil()
v2 = v.floor()
v2 = v.round()
v2 = v.sign()
v2 = v.allCloseToInt()

boolResult = v.any()  # boolResult is a logical scalar
boolResult = v.all()
boolResult = v.isBool()

v2 = v.logical_not()  # also accessible via not keyword

k = v.len()
k = v.nn()            # number of nulls
k = v.nnn()           # number of nonnulls

v = dg.ParVec.ones(size)
v = dg.ParVec.zeros(size)
v = dg.ParVec.broadcast(size, value)
v = dg.ParVec.range([start,] stop)
v2 = v.findInds()
```

```
m = v.max()
m = v.min()
m = v.sum()

m = v.norm(ord=1)     # 1-norm

v.randPerm()
```

Description

These methods have their standard Python or SciPy definitions, with the following exceptions.

Assignment ("=") of a `ParVec` to another variable name does not create a copy of the object, following Python usage for complex objects. The `copy` method can be used if a copy is needed.

The `allCloseToInt` method returns a Boolean scalar denoting whether all the elements of the vector are within machine precision of an integer value.

The `broadcast` method creates a vector of constant value, equivalent to (but faster than) `ones(size)*value`.

The `norm` method (modeled on the NumPy method) calculates only the 1-norm.

The `nn` and `nnn` methods return the number of nulls and nonnulls, respectively, with `nnn` behaving the same as SciPy's `getnnz`. The `findInds` method returns the indices of nonnull elements of the input `ParVec` instance, the same as NumPy/SciPy's `nonzero` method. The distinct names were chosen to provide future flexibility for a sparse `ParVec` class with a configurable null (zero) element.

The `randPerm` method randomly permutes the elements of the `ParVec` instance.

3.2.3 Built-in methods for SpParVec vectors

The `SpParVec` class implements data structures and operations for sparse distributed vectors; "sparse" in the sense that only non-null values are actually stored and represented in the vector.

Note: Operations on `SpParVec` instance(s) typically operate only on the non-null values, which is different from MATLAB behavior. Thus the `all` method, for instance, returns True if all the nonnull elements are True, ignoring the values of any null elements. Similarly, subtraction of a scalar will subtract the scalar value only from the nonnull elements of the vector. These semantics are intended to reduce the likelihood of sparse vectors unintentionally becoming much larger.

Version 0.1

The `SpParVec` class represents sparse vertex and directed-edge vectors. Its implementation for distributed memory includes a significant but not complete set of methods, which are mapped onto standard Python operators or methods consistent with clear understandability.

Note: In version 0.1, SpParVec instances only support a 64-bit floating-point or integer elemental datatype, and no explicit Boolean datatype. Any SpParVec whose nonnull elements are each either 0 or 1, such as "sparsevec < k", will be viewed as a Boolean vector for indexing purposes. The isBool() method tests this characteristic.

Syntax

```
# v, v2, and v3 are SpParVec instances
# k is an integer scalar, m is an integer or floating-point scalar
v2 = v > k            # v2 is a Boolean vector, where each element
v2 = v >= k           #  is the comparison of the corresponding
v2 = v < k            #  element of v with the scalar k
v2 = v <= k
v2 = v == k
v2 = v != k
v2 = ~v               # negation (only for Boolean vectors)
v3 = v & v2

v3 = v + v2
v += v2
v3 = -v
v3 = v – v2
v -= v2
v3 = v * v9           # only for v9 a ParVec instance
v3 = v / v9           # only for v9 a ParVec instance
v3 = v % v9           # only for v9 a ParVec instance

m = v[k]
v2 = v[vec]           # vec an integer SpParVec
v[k] = m
v[vec] = m            # see below
v[v2] = v3

del v[k]
del v[vec]            # vec an integer ParVec

v.abs()
v2 = abs(v)
```

Description

These methods have their standard definitions in Python with the following exceptions:

- Only a key set of indexing ("[]") modes are supported. [[**FIX**: update]]
    - Indexing on the right-hand side of an equation by a scalar

- Indexing on the right-hand side of an equation by a non-Boolean ParVec instance
- Indexing on the left-hand side by a scalar index (key) with a scalar value
- Indexing on the left-hand side by a Boolean ParVec index with a scalar value
- Indexing on the left-hand side by a Boolean ParVec index vector with a ParVec value vector the same length as the SpParVec instance being modified. Only the value elements corresponding to the True elements in the index vector are referenced.

3.2.4 Non-built-in methods for SpParVec vectors

The ParVec class implements several other methods, some of which are common Python names and some which are not.

Syntax

```
# v and v2 are SpParVec instances, m a scalar
v2 = v.copy()         # deep copy
vDense = v.toParVec() # converts to a (dense) ParVec

boolResult = v.any()  # boolResult is a logical scalar
boolResult = v.all()
boolResult = v.isBool()

v2 = v.logical_not()  # also accessible via not keyword

k = v.len()           #
k = v.nn()            # number of nulls
k = v.nnn()           # number of nonnulls

v = dg.SpParVec(size)      # all nulls
v = dg.SpParVec.ones(size) # all ones
v = dg.SpParVec.range([start, ]stop))
v.spones()                 # all non-nulls are set to 1
v.set(m)                   # all non-nulls are set to m
v.sprange()                # all non-nulls are set to their index

m = v.sum()
```

Description

These methods have their standard Python or SciPy definitions, with the following exceptions.

Assignment (“=”) of a SpParVec object to another variable name does not create a copy of the object, following Python usage for complex objects. The copy method can be used if a copy is needed.

The `len` method returns the length (*i.e.*, the maximum number of non-nulls there could ever be) of the `SpParVec` object. The `nn` and `nnn` methods return the number of nulls and nonnulls, respectively, with `nnn` behaving the same as SciPy's `getnnz`. The distinct names were chosen to provide future flexibility for the `SpParVec` class' null (zero) element to be configurable.

The `SpParVec` constructor itself creates an instance of the length specified with all elements null. The `ones` method creates a `SpParVec` instance of the length specified with all elements set to 1. The `range` method creates a `SpParVec` instance of the length specified with each element set to its index in the vector, with the optional `start` argument as in standard Python. The `spones` method modifies the bound `SpParVec` instance by setting all its nonnull elements to 1.0. The `set` method modifies the bound `SpParVec` instance by setting all its nonnull elements to the passed value. The `sprange` method modifies the bound `SpParVec` instance by setting each of its nonnull elements to its index in the vector.

The `sum` method sums the nonnull elements of the bound `SpParVec` instance, returning the scalar result.

3.2.5 Built-in methods for DiGraph objects

The `DiGraph` class represents directed graphs. Its implementation for distributed memory includes a small set of methods, some of which are mapped onto standard Python operators or methods consistent with clear understandability.

Version 0.1

`DiGraph` instances only support a 64-bit integer elemental datatype, and no explicit Boolean datatype.

Syntax

```
# G1, G2 and G3 are DiGraphs
G3 = G1 + G2          # elemental addition
G3 += G2              # elemental addition in place
G3 = G1 * G2          # elemental multiplication
G3 *= G2              # elemental multiplication in place
G3 = G1 / G2          # elemental division
G3 /= G2              # elemental division in place

print G
```

Description

These methods have their obvious definitions, with the caveat that the multiplication and division operators yield a null (zero) value in any element for which at least one of the graphs has a null entry. Because `DiGraph` are often large objects, in-place operators may often make sense to conserve the amount of memory consumed by temporary or transient objects.

The `print` method for `DiGraph` objects can be problematic, as they can often be extremely large (billions of elements), for which text display is rarely useful. For v0.1, for small graphs (defined as 100 edges or fewer), the print method will call the `toParVec` method and print those results. For larger graphs, you can accomplish this yourself by invoking the `toParVec` method manually and printing those results.

3.2.6 Non-built-in methods for DiGraph objects

In addition to built-in methods, a number of other utility methods are implemented for the `DiGraph` class.

3.2.6.1 Simple methods

Syntax

```
# G and G2 are DiGraphs
G2 = G.copy()          # deep copy
k = G.nvert()          # number of vertices
k = G.nedge()          # number of edges
G.ones()               # set all non-null edge-weights to 1.0
G.bool()               # set all non-null edge-weights to True
G.set(k)               # set all non-null edge-weights to k
G.mulNot(G2)           # see below
```

Description

These methods have their obvious definitions with the following elaborations.

Assignment (“=”) of a `DiGraph` object to another variable name does not create a copy of the object, following Python usage for complex objects. The `copy` method can be used if a copy is needed. `mulNot` takes the logical inverse of each element of the second `DiGraph` before doing the multiplication. Because it does this elementally, there is no extra memory consumed by the inverse values.

3.2.6.2 toParVec

The `toParVec` method of the `DiGraph` class decomposes a `DiGraph` instance to its edges .

Syntax

```
[source, dest, weight] = G.toParVec()
```

Description

The `toParVec` method of the `DiGraph` class decomposes a `DiGraph` instance to its edges, returning `ParVec` instances containing the source vertices, destination vertices, and edge-weights, respectively. Other than the bound `DiGraph` instance, there are no input parameters.

The `toParVec` method is almost the converse of the `DiGraph` method, with the difference that, since the `DiGraph` method sums duplicate edges, the output of `toParVec` may have fewer edges, though the same set of vertices.

3.2.6.3 reverseEdges

The `reverseEdges` method of the `DiGraph` class reverses the direction of each edge of a `DiGraph` instance .

Syntax

```
G.reverseEdges()
```

Description

The `reverseEdges` method of the `DiGraph` class reverses the direction of each edge of a `DiGraph` instance. Other than the bound `DiGraph` instance, there are no input parameters. The method works on the DiGraph instance in place, so destroys its input contents.

3.2.6.4 subgraph

The `subgraph` method of the `DiGraph` class creates a new graph from a selected set of vertices of an existing `DiGraph` object and those vertices' incident edges.

Syntax

```
G2 = G1.reverseEdges(vertrange[, vertrange2])
```

Description

The `subgraph` method of the `DiGraph` class creates a new `DiGraph` instance by selecting a set of vertices of an existing `DiGraph` instance and all the edges incident to those vertices. The required input argument `vertrange` specifies a range (*i.e.*, a consecutive set of vertices) of vertices (and out-edges incident to them) to be used for the new `DiGraph` instance. The optional input argument `vertrange2` specifies a distinct range of vertices (and in-edges incident to them) to be used for the new `DiGraph` instance. If `vertrange2` is not specified, `vertrange` designates both out- and in-vertices.

Example

The code below creates a new `DiGraph` from the first half of the vertices in G and edges whose source and destination are both one of those vertices.

```
G2 = G.subgraph(dg.DiGraph.range(G.nvert()/2))
```

The code below creates a new DiGraph with out-vertices of the first half of the vertices in G , in-vertices equal to the second half of the vertices in G, and edges whose source is in the first set and destination is in the second.

```
nvG = G.nvert()
G3 = G.subgraph(dg.DiGraph.range(nvG/2), dg.DiGraph.range(nvG/2, nvG))
```

3.2.6.5 degree / sum / max / min

The degree, sum, max, and min methods of the DiGraph class calculate, respectively, the degree (count), sum, maximum, and minimum edge-weight of the edges of each vertex of a DiGraph instance.

Syntax

```
inoutdegs = G.degree([dir=dg.InOut])
inoutsums = G.sum([dir=dg.InOut])
inoutmaxs = G.max([dir=dg.InOut])
inoutmins = G.min([dir=dg.InOut])
```

Description

The sum, max, and min methods of the DiGraph class calculate, respectively, the sum, maximum, and minimum edge-weights of the edges of each vertex of a DiGraph instance, returning a ParVec object. The optional dir argument specifies whether the operation is performed on InOut (default), In, or Out edges.

Example

The code below calculates the sum of the edge-weights of the out-edges of the vertices of a DiGraph.

```
outmaxs = G.max(dg.DiGraph.out)
```

3.2.6.6 scale

The scale method of the DiGraph class multiplies the edge weights of each vertex of a DiGraph instance by the corresponding element of a vector of scale factors.

Syntax

```
G.scale(scaleV[, dir=dg.Out])
```

Description

The scale method of the DiGraph class multiplies the edge weights of each vertex of a DiGraph instance by the corresponding element of a SpParVec vector of scale factors. The optional dir argument specifies whether the operation is performed on Out (default) or In edges.

Example

The code below normalizes the out-edge weights of each vertex of a `DiGraph` instance such that the sum of the edge-weights of each vertex is 1.0.

```
scalefac = dg.ParVec.ones(G.nvert()) / G.sum()
G.scale(scalefac)
```

3.2.7 Advanced methods for DiGraph objects

The `DiGraph` class implements several advanced methods.

3.2.7.1 neighbor

The `neighbor` method of the `DiGraph` class calculates the neighbors of a set of starting vertices in a `DiGraph` instance.

Syntax

```
neighbors = G.neighbor(start[, nhop=1, sym=False])
```

Description

The `neighbor` method of the `DiGraph` class calculates, for a `ParVec` of input starting vertices, which vertices are neighbors; *i.e.*, connected by out-edges. The optional `nhop` argument determines how many hops from the starting vertices are used to calculate the neighbors; the default is 1. The optional expert argument `sym` denotes whether the graph is a symmetric graph, which if True enables the operation to run faster. The return value is a `ParVec` with the indices of neighboring vertices.

Example

If `start` is a Boolean `ParVec` of starting vertices, the call below will return all vertices connected via out-bound edges within one hop.

```
neighbors = G.neighbor(start)
```

3.2.7.2 pathsHop

The `pathsHop` method of the `DiGraph` class calculates the vertices that can be reached from a set of starting vertices in a `DiGraph` instance, also returning which of the start vertices has an edge to each new vertex. `pathsHop` is equivalent to one step in a breadth-first search.

Syntax

```
[fromV, toV] = G.pathsHop(start[, sym=False])
```

Description

For a `ParVec` of input starting vertices, the `pathsHop` method calculates which vertices can be reached across a single edge in the `DiGraph` instance and, for each reachable vertex, which starting vertex has an edge to it. The source vertices and the new vertices are returned in `ParVec` and Boolean

ParVec instances, respectively. The set of reachable vertices may include starting vertices. In the case of more than one start vertex having an edge to a reachable vertex, the highest-numbered vertex is selected. pathsHop can be used repeatedly to implement a breadth-first search. The optional expert argument sym denotes whether the graph is a symmetric graph, which if True enables the operation to run faster.

Example

If start is a Boolean ParVec of starting vertices, the code below finds all reachable vertices with the call, and then selects just the newly found vertices from the set of reachable vertices.

```
[fromV, toV] = G.pathsHop(start)
newV = toV & ~start
```

3.2.8 DiGraph

The DiGraph method creates a directed graph from the edges passed to it.

Syntax

```
G = dg.DiGraph(source, dest, weight, nVertOut[, nVertIn])
G = dg.DiGraph()
```

Description

The DiGraph method creates a DiGraph instance. The required input parameters source and dest are ParVec objects created by the program or by generators such as genGraph500Edges. The required input parameter weight can be a ParVec or a scalar. The required input parameter nVertOut is an integer defining the number of vertices that have out edges in the graph. The optional input parameter nVertIn is an integer defining the number of vertices that have in edges in the graph; if all vertices potentially have both in and out edges, nVertIn may be omitted. The output argument is a DiGraph object. The values of any duplicate edges (same source and destination) are summed in the creation of the DiGraph object. The DiGraph method is almost the converse of the toParVec method, with the difference that, since the DiGraph method sums duplicate edges, the output of toParVec may have fewer edges, though the same set of vertices.

The alternate form of calling the DiGraph method with no argument creates a DiGraph instance with an empty underlying graph object. This is useful for certain constructors, like genGraph500Edges, which populate the underlying graph object themselves.

DiGraph implements the functionality of Kernel 1 of the Graph500 benchmark.

Example

The code below creates a star graph with N vertices, with directed edges of weight 1 going only from vertex 0 to all vertices (including vertex 0).

```
source = dg.ParVec.zeros(N)
dest = dg.ParVec.range(N)
weight = dg.ParVec.ones(N)
G = dg.DiGraph(source, dest, weight, N)
```

3.3 Package and class structure

KDT is structured as shown in Table 1, with some methods omitted for brevity. The complete list of functions can be seen by executing (within IPython)

```
        import DiGraph as dg
        dir(dg)
        dir(dg.DiGraph)
        dir(dg.ParVec)
        dir(dg.SpParVec)
```

Entity	Name	Elements	Comments
Module	Graph		
Class	ParVec		Distributed parallel vector
		ParVec, [], =, +, -, += , -=, <>, >, findInds, abs, any, all, nnn, ...	Methods
Class	SpParVec		Distributed parallel sparse vector
		SpParVec, [], =, +, -, += , -=, <>, >, findInds, abs, any, all, nnn, ...	Methods
Module	DiGraph		
Class	DiGraph		Directed graph
		DiGraph, genGraph500Edges, load, fullyConnected	Constructors
		centrality, cluster	Algorithms
		bfsTree, isBfsTree, neighbors, pathsHop, toParVec, reverseEdges , subgraph	Graph primitives
		load	I/O
		[], nvert, nedge, degree, +, *, ones, set	General-purpose routines
		InOut, In, Out	Constants
Class	ParVec		Inherited from Graph
	SpParVec		Inherited from Graph

Table 1. KDT library hierarchy

3.4 Graph Generators

The `DiGraph` class includes the graph generators in this section. With v0.1, edges must be created as an `ParVec` object directly, either by the `load` method, a constructor such as `fullyConnected`, or by using an application-specific method like `genGraph500Edges`.

3.4.1 genGraph500Edges

The `genGraph500Edges` function creates a graph following the specifications for the V1.1 Graph500 benchmark's input graph [Graph500]. The edges are inserted into the `DiGraph` object passed and represent an RMAT graph with specific values provided by the benchmark.

Syntax

```
time = G.genGraph500Edges(scale)
```

Description

The `genGraph500Graph` method creates an input graph as defined by the Graph500 benchmark. The required input parameter `scale` (logarithm base 2 of the number of desired vertices) defines the number of vertices. The edges are directed, though each edge has a twin going in the other direction because the specification requires the graph to be symmetric. Some vertices may have no edges incident to them. The time returned from `genGraph500Edges` includes the execution time of converting the edge vector to a graph but does not include the time to create the edge vector, which is exactly the time measured by the Graph500 benchmark.

Example

The following code will creates a `DiGraph` instance and inserts edges into it that match the Graph500 specification, of size `scale`.

```
        G = dg.DiGraph()
        time = G.genGraph500Edges(scale)
```

3.4.2 `twoDTorus`

The `twoDTorus` method creates a `DiGraph` object reflecting the connectivity pattern of a 2D torus.

Syntax

```
G = dg.twoDTorus(nnode)
```

Description

The `twoDTorus` method creates a `DiGraph` with the connectivity pattern of a 2D torus; *i.e.*, connections to its north, west, south, and east neighbors, where the neighbor may be wrapped around to the other side of the torus. The required input parameter `nnodes` defines the number of nodes along one dimension of the torus; the torus and the graph representing it will have `nnodes**2` vertices. The newly created `DiGraph` object is the return value from the method. A 2D torus has an easily analyzed betweenness centrality value that can be useful for simple tests; specifically, each vertex of the `DiGraph` created by `twoDTorus(nnodes)` has an identical betweenness centrality value of 0.5 * 2**(3*`scale`*0.5) – 2**`scale` + 1 for an even `nnodes`.

Example

The following code creates a `DiGraph` instance G with `nnodes**2` vertices and inserts edges from every vertex to every other vertex, including itself.

```
        G = dg.DiGraph.twoDTorus(nnodes)
```

3.4.3 `fullyConnected`

The `fullyConnected` method creates a `DiGraph` object in which all vertices are directly connected to all other vertices. The edges are inserted into a newly created `DiGraph` object, which is the return value from the method.

Syntax

```
G = dg.fullyConnected(nvert)
```

Description

The `fullyConnected` method creates a `DiGraph`. with all vertices having a directed edge to all vertices, including itself. The required input parameter `nvert` defines the number of vertices. The newly created `DiGraph` object is the return value from the method.

Example

The following code creates a `DiGraph` instance G with `nvert` vertices and inserts edges from every vertex to every other vertex, including itself.

```
        G = dg.DiGraph.fullyConnected(nvert)
```

3.5 I/O

The `DiGraph` class includes the I/O-realted methods in this section.

3.5.1 `load`: Reading a graph from a file

A Matrix Market file can be loaded directly into a `DiGraph` object with the standard `load` I/O operation.

Syntax

```
G = dg.DiGraph.load(fname)
```

Description

The `load` method loads a file in the Coordinate Format of the Matrix Market Exchange Formats [MatrixMarket] directly into a `DiGraph` object.

Version 0.1

The source-vertex / destination-vertex information in the first two columns of the Matrix Market Exchange format is 1-based in the file, and is converted to 0-based in the load. With v0.1, the error-checking of a source-vertex or destination-vertex being out of bounds is not robust and can result in KDT aborting with Segmentation Faults or malloc errors.

Example

The following code will load the contents of the file `mymatrix.mtx` into a `DiGraph` instance named G.

```
G = dg.DiGraph.load('mymatrix.mtx')
```

3.5.2 `save`: Writing a graph into a file

A `DiGraph` object can be saved directly into a Matrix Market file with the standard `save` I/O operation.

Syntax

```
G.save(fname)
```

Description

The `save` method save a a `DiGraph` object directly into a file in the Coordinate Format of the Matrix Market Exchange Formats [MatrixMarket].

Example

The following code will save the contents of a `DiGraph` instance named G into the file `mymatrix.mtx`.

```
G.save('mymatrix.mtx')
```

3.5.3 `UFget`: Fetching a file from the University of Florida Sparse Matrix Library and loading it as a DiGraph

The `UFget` method downloads a named Matrix Market file from the University of Florida Sparse Matrix Library (link) and loads it into a `DiGraph` instance.

Syntax

```
G.UFget(fname)
```

Description

The `UFget` method downloads a file from the UF Sparse Matrix Library, untars the contents, and loads the contained file (in the Coordinate Format of the Matrix Market Exchange Formats) directly into a `DiGraph` instance.

Example

The following code will load the contents of the file `Andrianov/ex3sta1.tar.gz` from the UF Sparse Matrix Library into a `DiGraph` instance named G.

```
G = dg.DiGraph.UFget('Andrianov/ex3sta1')
```

3.5.4 `master`: Avoiding redundant print output

When running in parallel, each process will execute the Python code, including any `print` statements. For user output, this can lead to numerous copies of the output, often intermingled so as to make the output unintelligible. The `master` method is useful for avoiding this situation.

Syntax

```
bool = dg.master()
```

Description

The `master` method returns a Boolean result that is True in the master process of the underlying infrastructure and False in all other processes.

Example

The following code, taken from the `Graph500` method in KDT, restricts printing output just to the master process.

```
        if dg.master():
            print 'Graph500 benchmark run for scale = %2i' % scale
            print 'Kernel 1 time = %8.4f seconds' % K1elapsed
            print 'Kernel 2 time = %8.4f seconds' % K2elapsed
```

3.6 Toolbox Structure

KDT will eventually have multiple implementations, targeting serial and distributed-parallel versions both memory-based and disk-based, as reflected in Table 2 below. The serial versions are intended for program development and solving small problems, while the distributed versions provide unique value. Only the distributed in-memory version (kdtdm) is implemented in v0.1; all other components are shown in grey font.

	Module	Class	Methods	Comments
sm	<not expanded here, contents superset of kdtdm>			Serial, in-memory
dm				Distributed, in-memory
	Graph			
		Graph		General Graph
		ParVec		Vertex vector
			len, degree, …	Methods
	DiGraph			
		DiGraph		Directed graph
			central, cluster, indegree, …	Methods
	MultiGraph			Future Multigraph
	HyperGraph			Future Hypergraph
sd	<not expanded here, contents roughly same as kdtdm>			Future Serial, on disk
dd	<not expanded here, contents roughly same as kdtdm>			Future Distributed, on

		disk

Table 2 Diverse KDT implementations and graph types

We anticipate multiple implementations of the KDT interface, adhering to this structure and naming for all functions that a particular implementation supports. The current distributed in-memory functionality of KDT is imported as

```
        import DiGraph as dg
```

Anticipated future versions may effect `import` statements.

3.7 Advanced Usage

KDT v0.1 supports only graphs with a single edge between any two vertices, which doesn't address nearly all graph types. While fully general graphs will have to wait for future releases, v0.1 does support (via built-in Python mechanisms) multiple graphs of different types of data. For instance, one potential KDT use case is to analyze data from protein-protein, DNA-protein, and other interactions. Data about a particular organism (*e.g., Shewanella*) could be collected in a single `DiGraph` instance which has other `DiGraph` instances as members, as portrayed by the following code.

```
shew = dg.DiGraph()
shew.prot2prot = dg.DiGraph.load('shewanella/protein_protein.mtx')
shew.DNA2prot = dg.DiGraph.load('shewanella/DNA_protein.mtx')
d2p_bc = shew.DNA2prot.centrality('approxBC')
p2p_bc = shew.prot2prot.centrality('approxBC')
d2p_deg = shew.DNA2prot.degree()
p2p_deg = shew.prot2prot.degree()
```

In this example, each graph is still operated on independently, but where graph operations make scientific or mathematical sense, `DiGraph` graphs can be joined together.

4 Practicalities

4.1 Downloading

[[FIX: include details.]]

4.2 Installation

[[FIX: include details: MPI version, IPython version, (SWIG version?).]]

4.3 Execution (Python/ IPython)

[[FIX: include details: MPI version, IPython version, (SWIG version?).]]

4.3.1 IPython

KDT has been used interactively only with IPython. The command-line and IPython commands to invoke that follow (same as section 2).

```
[sam@neumann ~]$ ipython
```

```
Python 2.4.3 (#1, Nov 11 2010, 13:30:19)
Type "copyright", "credits" or "license" for more information.

IPython 0.8.4 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints
more.
```

[[ToDo: double-check path]]

```
In [1]: import Graph500
Activating auto-logging. Current session state plus future input
saved.
Filename       : .KDT_log    #This output is from the startup of the IPython logstart
Mode           : over        # mechanism used for the sendFeedback function
Output logging : False       # from Section 1.5
Raw input log  : False       #      ""
Timestamping   : False       #      ""
State          : active      #      ""

Using fully connected graph generator
Graph500 benchmark run for scale =  8
Kernel 1 time =   0.0001 seconds
Kernel 2 time =   0.6300 seconds
                      0.0098 seconds for each of 64 starts
Kernel 2 TEPS = 6.6576e+06

In [2]:
```

4.3.2 Python

KDT has been used in parallel only with the Python language processor, namely Python 2.4.3. The MPI used was OpenMPI 1.2.2. The Linux version was 2.6.18.

For parallel execution with Python and MPI,you can use the following command-line. [ToDo: double-check once directories are moved.]

```
[sam@neumann test]$ mpirun -n 4 python Graph500.py
Using Graph500 graph generator
Duplicates removed: 1891 and self-loops removed: 0
Duplicates removed: 1891 and self-loops removed: 20
Graph500 benchmark run for scale =  8
Kernel 1 time =   0.0011 seconds
Kernel 2 time =   0.0400 seconds
                      0.0006 seconds for each of 64 starts
Kernel 2 TEPS = 6.9920e+06
[sam@neumann test]$
```

Appendix A. Implementing Graph Algorithms with the Combinatorial BLAS

The Combinatorial BLAS [Buluc] is described by its authors as:

> We describe the Parallel Combinatorial BLAS, which consists of a small but powerful set of linear algebra primitives specifically targeting graph and data mining applications. We provide an extendible library interface and some guiding principles for future development. The library is evaluated using two important graph algorithms, in terms of both performance and ease-ofuse. The scalability and raw performance of the example applications, using the combinatorial BLAS, are unprecedented on distributed memory clusters.

It provides C++ interfaces that are appropriate for many HPC users, but no interface from the high-level productivity languages such as Python and the M language of MATLAB™. KDT provides such an interface via Python, but the current KDT exposes just a small fraction of the power of the Combinatorial BLAS. This section describes the implementation of graph algorithms with the Combinatorial BLAS' primitives, and describes a few of the Combinatorial BLAS primitives whose full power is not yet exposed. The Combinatorial BLAS' basic structure is a sparse matrix, which KDT uses with From vertices as rows and To vertices as columns. The Combinatorial BLAS' C++ interfaces are wrapped into Python as the `pyCombBLAS` module, with classes `pyDenseParVec`, `pySpParVec`, and `pySpParMat` that correspond directly with KDT's `ParVec`, `SpParVec`, and `DiGraph` classes.

This information is provided so that KDT users can understand the generality of what could be implemented eventually, and so that heroic KDT users can implement their own extensions to KDT by using the `pyCombBLAS` directly.

A.1 Implementing Breadth-first Search with the Combinatorial BLAS

One of the core algorithms implemented in KDT v0.1 is creating a breadth-first search tree from a graph and a root vertex. The code for this is found in `DiGraph.bfsTree`.

The abstract BFS algorithm starts with a "fringe" or "frontier" of vertices that has been first reached in the prior step of the algorithm, and in each step calculates all as-yet-unreached vertices that are reachable from the fringe and makes those vertices the new fringe. In sparse-matrix terms, the graph is the dual of the adjacency matrix, with destination vertices corresponding to rows and source vertices corresponding to columns as shown below. A standard sparse-matrix/vector multiplication multiplies corresponding elements of a row of the matrix and the vector and then adds those products. The operation needed for BFS is similar but different. We still want to identify positions where both the matrix and the vector have nonnulls, but instead of a multiplication the result just needs to be the position of the nonnull in the row/column (denoting which fringe vertex is the parent of the new vertex in the BFS tree), and instead of addition of multiple nonnulls in the row/column we just need to select one of them (*i.e.*, if a new vertex is reachable from multiple vertices in the fringe, in general it doesn't matter which one of those vertices is denoted as the parent in the BFS tree). The Combinatorial BLAS and KDT draw on the rich theory of linear algebra on semirings [Gilbert, p. 28-31] to implement this operation. Using the same computational structure and communication pattern but replacing the multiplication and addition of standard matrix-vector multiplication with selection (of the column of the matrix element) and maximum, BFS achieves the needed computation.

Let's consider the example shown in **Error! Reference source not found.**, with a graph and its dual adjacency matrix, which is transposed to work properly with matrix-vector multiplication.

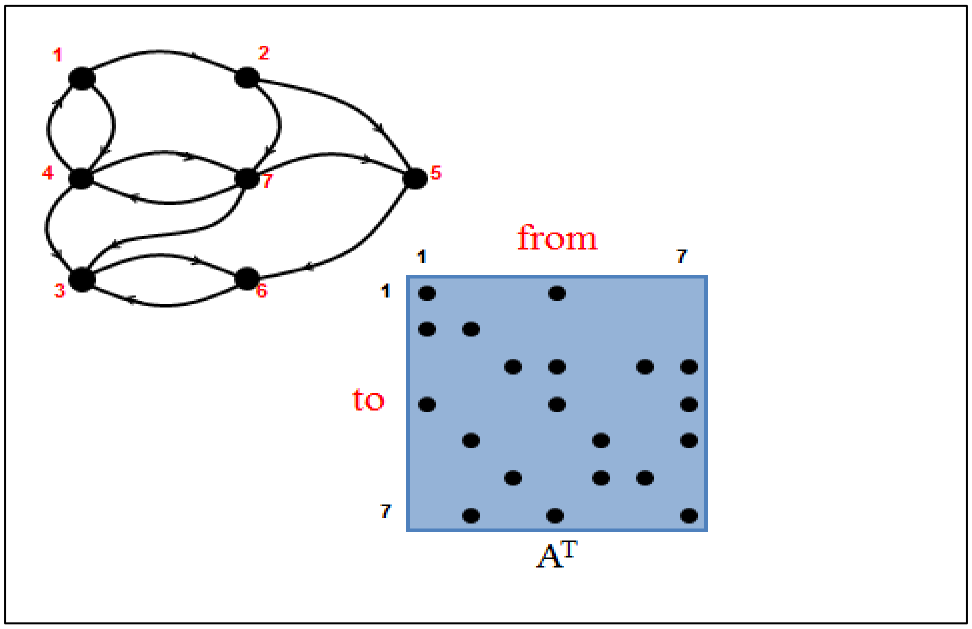

Figure 1. Graph and adjacency matrix example.

Let's make vertex 1 the root of the BFS tree and take the first step of the algorithm in Figure 2. The fringe vector has a nonnull only in position 1. Since the "multiplication" of row 2 of the matrix with the fringe vector results in a nonzero in position 1, element 2 of the result matrix is set nonzero, with its value equal to the value of the nonzero in the vector "product". More precisely, the algorithm selects any position with nonzeros in both in the row and the fringe vector, with the result being the value of the fringe vector element (which is previously set to its position in the element). Applying this algorithm, vertices 2 and 4 are calculated as the next level of the BFS tree (with vertex 1 as each's parent), and the red edges are added to the BFS tree. (The algorithm truncates from the new fringe any vertices that have already been visited, as with the self-loop from vertex 1 to itself.)

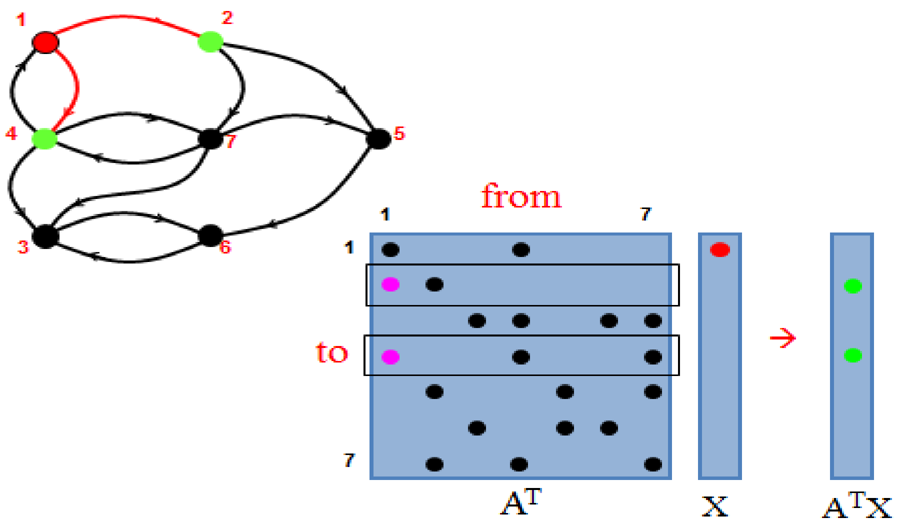

Figure 2. First step of BFS algorithm

For the next step, the result column vector of the previous step is used as the fringe vector for the current step, with each nonnull element set to its position in the fringe. As illustrated in Figure 3, the same algorithm is applied. The calculation of the edge to vertex 7 illustrates the maximum step (in place of the addition in standard matrix-vector multiplication). Vertices 2 and 4 both have edges to vertex 7. The algorithm chooses the maximum-numbered vertex (vertex 4, denoted by the dark circle around the pink dot in the matrix) as the parent. Vertices 3, 5, and 7 are in the new fringe with parents of 4, 2, and 4, respectively. The resulting green edges are added to the BFS tree.

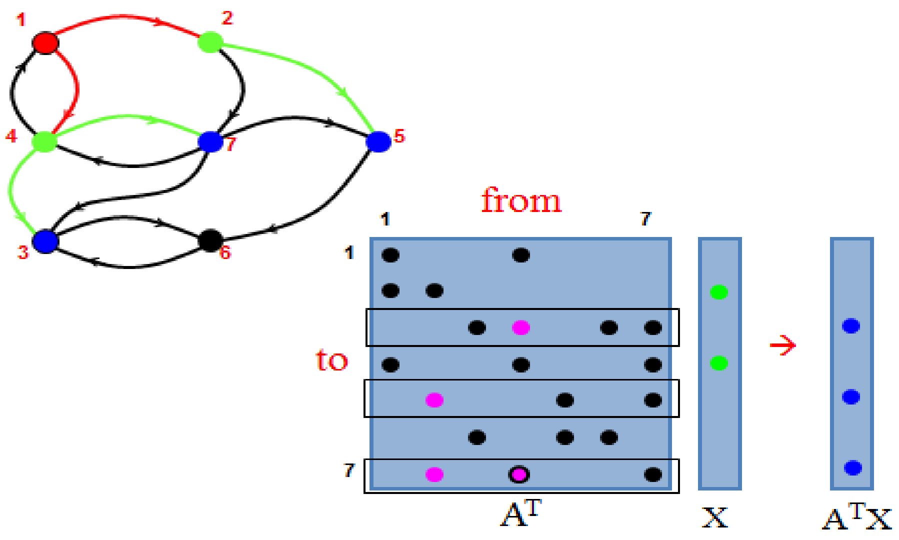

Figure 3. Second step of BFS algorithm.

The next-to-last step of the algorithm for this graph is illustrated in Figure 4. The only as-yet-unvisited vertex is vertex 6, which is reachable from either vertex 3 or 5. The tie-breaker (maximum vertex number) determines that vertex 6's parent is vertex 5, and the blue edge is added to the BFS tree.

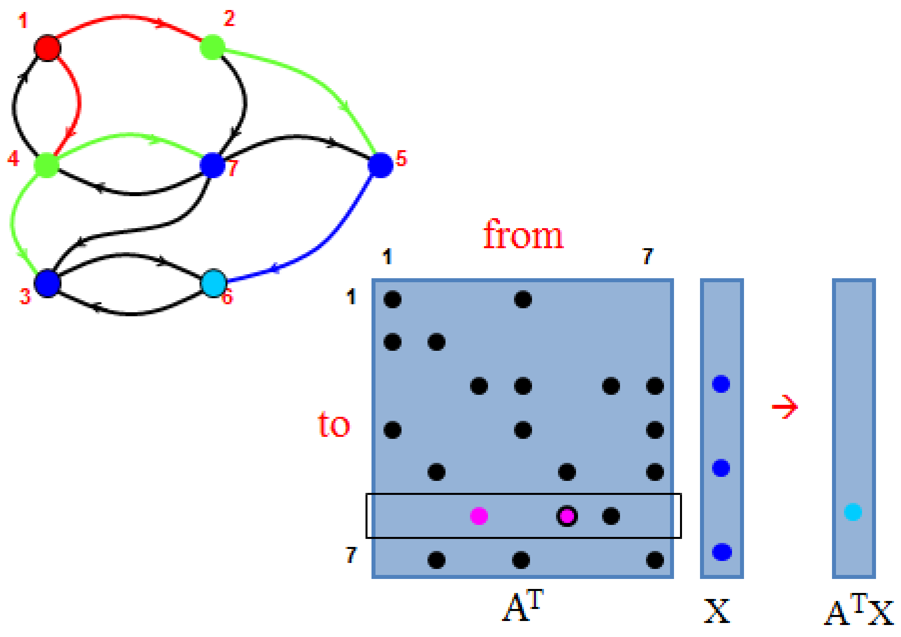

Figure 4. Third step of BFS algorithm.

The final step of the algorithm uses the result vector of the third step, with only vertex 6 in the fringe, and detects no unreached vertices, at which point the algorithm terminates. The parent vector result is [1, 1, 4, 1, 2, 3, 4]; note that the root vertex is its own parent.

A.2 Powerful Operations in the Combinatorial BLAS

The Combinatorial BLAS has several operations which have powerful generality, of which sparse-matrix/vector multiplication on semirings, illustrated in the previous section, is one. The full complement of `pyCombBLAS` operations is specified in the file `pyCombBLAS.i` in the `kdt/pyCombBLAS` directory.

A.2.1 Sparse-matrix/vector multiplication on semirings

As illustrated in the previous section, replacing the element-wise multiplication and summation reduction of the standard sparse-matrix/vector multiplication operation with other semiring operations can perform powerful operations on the graph. The standard operation is available in `pyCombBLAS` as `SpMV_PlusTimes`. The operation used for the BFS tree calculation is available as `SpMV_SelMax`. [**FIX**: PlusTimes and SelMax order inconsistency may be fixed.] Eventually this operation will be generalized in the style of the reduction and apply operations described below.

A.2.2 Applying an elemental operator

`pyCombBLAS` has a variety of elemental operations built-in (see the file `pyCombBLAS.i` for a complete list), both unary (*e.g.*, `abs`) and binary (*e.g.*, `greater`, or `fmod` with a constant). These operations can be applied to each element of a `pySpParMat` object by the `pySpParMat.Apply` method.

A.2.3 Applying an elemental operator with a column-specific value

`pyCombBLAS` can also apply these elemental operations with a second input that is column-specific. The `pySpParMat.ColWiseApply` method takes a second argument (a `pySpParVec` instance), which provides the second element for a binary operation. For instance, the `DiGraph.scale` method, which multiplies each out-edge of a DiGraph by the corresponding element of a `SpParVec` , is implemented as

```
      self.spm.ColWiseApply(other.spv, pcb.multiplies())
```

A.2.4 Reducing the out-edges (columns) of a `DiGraph` (`pySpParMat`) with configurable operators

`pyCombBLAS` has a `pySpParMat.Reduce` method that is configurable with the same unary and binary operations. The `Reduce` method, in addition to its bound `pySpParMat` instance, accepts a dimension (columns or rows), a binary (reduction) function, and a unary function that's applied elementally before the binary function. For example, the column-wise sum of the absolute values of the nonnull elements could be implemented by

```
ret = self.spm.Reduce(pcb.pySpParMat.Column(),pcb.plus(), pcb.abs());
```

Appendix B. Glossary

The following terms are used with their given meanings throughout this document.

Graph: A collection of **vertices** and **edges** connecting the vertices.

Edge vector: A vector of tuples, with each tuple containing the indices of the vertices upon which an edge is incident and the weight of the edge. Represented by classes specific to the type of graph, *e.g.* `DiEdgeV`.

Vertex vector: A vector of integers, each being the index of a vertex being referred to. Represented by the `DiEdgeV` class generic to all types of graphs.

DiGraph: A collection of vertices and directed edges connecting the vertices.

Appendix C. References

[Bader] D. Bader, S. Kintali, K. Madduri, "Approximating betweenness centrality", The 5th Workshop on Algorithms and Models for the Web-Graph (WAW2007), San Diego, CA December 11-12, 2007.

[Buluc] Aydin Buluc and John Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications", Technical Report UCSB-CS-2010-18, UCSB Computer Science Department, Oct 2010. http://www.cs.ucsb.edu/research/tech_reports/reports/2010-18.pdf

[Freeman] L. Freeman, "A set of measures of centrality based on betweenness", Sociometry 40(1), 1977, 35-41.

[Gilbert] John Gilbert and Aydin Buluc, "Tools and Primitives for High Performance Graph Computation", SIAM Minisymposium on Analyzing Massive Real-World Graphs, July 2010, http://www.cs.ucsb.edu/~gilbert/talks/SIAMannual12july2010.pdf.

[Girvan] Girvan, M., and Newman, M.E.J. *Community structure in social and biological networks,* PNAS 99(12): 7821-7826 (2002).

[Graph500] http://www.graph500.org/

[MatrixMarket] http://math.nist.gov/MatrixMarket/formats.html

[SSCA] Synthetic Scalable Compact Applications, http://www.highproductivity.org.

[van Dongen] A.J. Enright, S. Van Dongen, C.A Ouzounis. *An efficient algorithm for large-scale detection of protein families*, Nucleic Acids Research 30(7):1575-1584 (2002).

Appendix D. Revision History

R0.01	December 16, 2010	Initial draft
R0.02	December 20, 2010	Revisions including comments from Aydin Buluc and Drew Waranis
R0.03	January 22, 2011	Bring current with code base (almost), documenting neighbors() and pathHop() as well as several other methods.
R0.04-05	February 12, 2011	Bring current with code base, documenting SpParVec class, many new ParVec and DiGraph methods, adding start-up and Implementing-graphs-on-CombBLAS information
R0.06	February 13, 2011	Finish Implementing-graphs-on-CombBLAS, add UFget, stipple cluster(), add DiGraph.save