The Knowledge Discovery Toolbox (Version 0.2) User Guide

Last change date : November 26, 2011 Revision : r0.13

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 0.2

This early release provides a small selection of functions on directed graphs ranging from simple exploratory functions to complex algorithms, and initial support for *semantic* graphs; *i.e.*, those with attributes on vertices and/or edges. 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 0.2***" 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 to large graphs. 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. KDT's goal is for the subject-matter expert (or her graduate student) to be able to compose KDT 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 (*e.g.*, each vertex of the graph created by `generate2DTorus(nnodes)` has an identical betweenness centrality value of 0.5 * 2**(3*`scale`*0.5) – 2**`scale` + 1 for an even `nnodes` , where scale is equal to log2(nnodes**2)).

1.2 Intended use cases

Version 0.2

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 Ranking the importance of Web pages

Analysis of the relative importance of Web pages led to the seminal PageRank algorithm, whose variations are the cornerstone of contemporary search engines.

1.2.3 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. With v0.2's semantic graph support, a graph with edges of multiple types can be analyzed for centrality on any type(s) of edges.

1.3 Deferred use cases

Version 0.2

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, bipartite graphs, multigraphs, and hypergraphs; matching or alignment; completely general-purpose attribute structures for vertices and edges; visualization of resulting graphs; support on other than x86-64 clusters; and a disk-based implementation, for problems that do not fit in the memory of a computational cluster.

1.4 Performance

KDT's goal is to enable quick and effective analysis of very large graphs, and thus performance is a vital aspect of KDT. Its performance has not yet been fully characterized. Performance of specific methods, where available, is provided in their descriptions.

Much of the early performance work with KDT has focused on the Graph500 benchmark, whose units are traversed-edges per second (TEPS). **On a 256-core cluster, the KDT implementation of Graph500 performs at over 1 GTEPS.** See section 5 for details.

1.5 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`.

1.6 Sending 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 then 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 = kdt.GenerateRMAT(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() method doesn't exist
In [47]: discG = G.deleteverts(bc.topK(10))
In [48]: kdt.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 graph from the input 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 illustrates 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 and that the environment variable KDTINSTALL has been set to the location of the unzipped/untarred package (see section 6.3.3 for details). You can type the following to start a serial Python 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 6.3.3.

```
[sam@neumann ~]$ which ipython
/usr/bin/python
[sam@neumann ~]$ ipython –debug $KDTINSTALL/examples/Graph500.py
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 1.6
Raw input log  : False      #      “”
Timestamping   : False      #      “”
State          : active     #      “”

Generating a Graph500 RMAT graph with 2^15 vertices...
Duplicates removed (or summed): 78842 and self-loops removed: 0
Generation took 0.328309s.
iteration 1: start=28166, BFS took 0.027983s, verification took
2.482376s and succeeded, TEPS=31,832,419
iteration 2: start=27666, BFS took 0.028234s, verification took
2.441357s and succeeded, TEPS=31,549,367
iteration 3: start=27587, BFS took 0.028846s, verification took
2.421630s and succeeded, TEPS=30,879,990
iteration 4: start=2117, BFS took 0.029042s, verification took
2.409176s and succeeded, TEPS=30,671,607
     […]
iteration 61: start=18643, BFS took 0.028643s, verification took
2.409366s and succeeded, TEPS=31,098,729
iteration 62: start=26284, BFS took 0.029493s, verification took
2.411485s and succeeded, TEPS=30,202,494
iteration 63: start=7687, BFS took 0.030046s, verification took
2.421176s and succeeded, TEPS=29,646,722
iteration 64: start=30285, BFS took 0.029399s, verification took
2.412161s and succeeded, TEPS=30,299,244
```

```
Graph500 benchmark run for scale = 15
Kernel 1 time =   0.3283 seconds

Kernel 2 BFS execution times
            min_time: 2.79829502105712891e-02
  firstquartile_time: 2.86501049995422363e-02
         median_time: 2.90445089340209961e-02
  thirdquartile_time: 2.93260216712951660e-02
            max_time: 3.00459861755371094e-02
           mean_time: 2.89925746619701385e-02
         stddev_time: 4.45555267017642357e-04

Kernel 2 number of edges traversed
            min_nedge: 8.90765000000000000e+05
  firstquartile_nedge: 8.90765000000000000e+05
         median_nedge: 8.90765000000000000e+05
  thirdquartile_nedge: 8.90765000000000000e+05
            max_nedge: 8.90765000000000000e+05
           mean_nedge: 8.90765000000000000e+05
         stddev_nedge: 0.00000000000000000e+00

Kernel 2 TEPS
            min_TEPS: 2.96467220212343894e+07
  firstquartile_TEPS: 3.03745608150749356e+07
         median_TEPS: 3.06689642658791840e+07
  thirdquartile_TEPS: 3.10911601670094803e+07
            max_TEPS: 3.18324191444078088e+07
  harmonic_mean_TEPS: 3.07239012190395705e+07
harmonic_stddev_TEPS: 1.69827840156330167e+04
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 [5]: parents
Out[2]: Limiting print-out to first 100 elements
Elements stored on proc 0: {25802,1,1,19693,32686,624,21394,27838,
28156,-1,14783,8652,-1,-1,26729,-1,30564,29617,-1,-1,19294,-1,
6541,26552,23513,31138,2797,2567,30313,32502,-1,-1,8402,-1,32686,-1,
21311,30589,-1,16940,23819,-1,28403,32686,21906,25699,32645,30794,
16244,28393,30280,8146,16578,23171,23683,31583,26762,29294,-1,32010,
32686,27230,30280,-1,-1,28311,27501,-1,30589,30280,32736,-1,31834,
12570,31009,14818,28403,30280,5489,31834,27838,-1,-1,31280,-1,-1,
31009,9680,23819,21028,28403,16244,-1,-1,30279,30280,32100,32153,
21906,32700,}
```

Looking at the code in `$KDTINSTALL/examples/Graph500.py`, we can see how the Graph500 specification is implemented with KDT methods.

```
import time
import numpy as np
import scipy as sc
import kdt
#
#          […]
#
#     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**; overridden Python operators are not highlighted.

```
# find the vertices of degree > 2
deg3verts = (G.degree(dir=kdt.DiGraph.InOut) > 2).findInds()
nstarts = 64                        # #times to create a BFS
# randomly pick some of those vertices as start points
deg3verts.randPerm()
starts = deg3verts[kdt.Vec.range(nstarts)]
```

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 start in starts:
     start = int(start)
     before = time.time()

     # the actual BFS
     parents = G.bfsTree(start)

     itertime = time.time() - before
     nedges = len((parents[origI] != -1).find())

     K2elapsed.append(itertime)
     K2edges.append(nedges)
     K2TEPS.append(nedges/itertime)

     i += 1
     verifyInitTime = time.time()
     verifyResult = "succeeded"
     if not k2Validate(G, start, parents):
          verifyResult = "FAILED"
     verifyTime = time.time() - verifyInitTime
```

```
    if kdt.master():
        print "iteration %d: start=%d, BFS took %fs, verification took %fs and %s, TEPS=%s"%(i, start, (itertime), verifyTime, verifyResult, splitthousands(nedges/itertime))
```

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. It achieves this by building a new `DiGraph` instance containing just the edges in the tree, validating in the process that no tree edge has the root as its destination. (Note that the `isBfsTree` code can be viewed in its installed position (usually in `/usr/lib64/python2.4/site-packages/kdt/Algorithms.py`) and also in `$KDTINSTALL/kdt/Algorithms.py`, for a Python 2.4 installation).

```
    tmp2 = parents != Vec.range(nvertG)
    treeEdges = (parents != -1) & tmp2
    treeI = parents[treeEdges.findInds()]
    treeJ = Vec.range(nvertG)[treeEdges.findInds()]
    # root cannot be destination of any tree edge
    if (treeJ == root).any():
        ret = -1
        return ret
    # note treeJ/treeI reversed, so builtGT has reversed edges, as
    #   needed by SpMV
    builtGT = DiGraph(treeJ, treeI, 1, nvertG)
```

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 Semantic Graph Example

[[**FIX**: add example from ICASSP paper]]

4 Algorithms, Methods and Classes

Version 0.2

The v0.2 release of KDT supports only directed graphs with single edges between any two vertices, via the `DiGraph` class.

Collections of vertices and edges are represented as `Vec` class instances. See section 4.3.9.6 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 kdt
```

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

4.1 Execution Model

4.1.1 Data Distribution and Parallel Execution

KDT's objects, such as `DiGraph` or `Vec` instances, are *global* and *distributed*; global meaning that a variable (say a graph `G`) will have the same name and meaning on all the cluster nodes participating in the computation, and that operations on that variable will operate on the whole variable, and distributed meaning that they are (in general) distributed across the memory of all the cluster nodes participating in the computation.

KDT supports parallel execution via the single-program-multiple-data (SPMD) model, meaning that every core runs its own copy of the Python/KDT program and communicates (within KDT) with the other cores as appropriate. The constraint this places on the KDT user is that the arguments passed to KDT functions must be the same on all cores. Most times this is trivial, as when an argument is the name of a variable. One place where this may require thought is in the use of random numbers, where having each core calculate its own random numbers locally may result in different sets of random numbers on each core. The `Vec.randPerm` function is recommended for ensuring random numbers are the same on all cores; the `examples/Graph500.py` file has an example of its use.

The algorithms and methods below by default execute across all vertices and edges in a graph, but can be constrained to execute only on vertices and/or edges whose attributes meet user-specified criteria, as described in the following section.

4.1.2 Semantic Graph Overview: Operations and Filters

The descriptions of graph algorithms and methods so far have used graphs that are homogeneous in their vertices and edges. Real-world data often includes vertices and edges with complex definitions with multiple attributes such as values, types, and time-stamps; so-called *semantic* graphs. For instance, in a social network, people communicate via cell-phone calls, text and email messages, and face-to-face encounters. Each of these different modes of communication could have its own type of edge in a KDT graph, and the methods can be customized to consider only certain type(s) of vertices or edges during a calculation. Because `Vec` instances are used for vertex information, similar filter operations are usable on `Vec` instances as for `DiGraph` instances, with the same data and execution models and customizability.

4.1.2.1 Execution model

The execution model for semantic `DiGraph` and `Vec` instances are the same as non-semantic `DiGraph` and `Vec` instances, though edges and vertices, respectively, are more richly defined. To use this extra information, users create *filters* that describe which vertices/edges should be used in subsequent operations on that instance. A filter is a Python function that looks at the contents of a vertex (edge) object and returns a True if the vertex (edge) is to be used and False otherwise. [[**FIX**: say more about what functions a filter can call?]] **When a filter is added to a graph or vertex-vector it will determine which vertices or edges are used for all operations until the filter is removed**. Multiple filters can be added to a graph or vector; they can be viewed as being run in series within each operation, before the core operation itself, so that only vertices/edges that pass (*i.e.,* return True from) all filters will be used in the operation. For instance, a `DiGraph` instance representing a social network could have edge-types representing cell-phone calls, text messages, and face-to-face encounters, and have a filter attached that selects just cell-phone and text-message edge types. Executing the `degree` method on this `DiGraph` instance would silently ignore edges of other types while calculating the degree of each vertex. With this execution model, the KDT developers expect that materializing the filtered data (and consuming the potentially large amount of memory of the graph or vector copy) can usually be avoided.

A few operations (*e.g.,* `find` and `reduce`) accept a `pred` (predicate) or `uniOp` (unary operator) argument that can be viewed as a temporary filter. A predicate must be functionally the same – a Python function that accepts a single element (edge or vertex) and returns a Boolean result of whether to use or ignore the element in the following calculation.

4.1.2.2 Data model

The default `DiGraph` and `Vec` objects have an elemental value that is a double-precision (*i.e.,* 64-bit) floating point value; *i.e.,* a length-300 `Vec` object has 300 64-bit doubles in it. A semantic graph needs to support more data about each element of the graph or vector, such as type, multiple values, start-time and end-time, etc. Balancing the needs for customizability and performance for semantic graphs, KDT supports two built-in object (structure) types (`Obj1` and `Obj2`), making it possible to use one object-type for edges and the other for vertices (though there's nothing intrinsic to the object-types that requires that). Creation of semantic `DiGraph` and `Vec` instances typically requires use of the `element` argument to a constructor method. For instance, to create a `Vec` instance to be used as a vertex vector, say with an elemental type of `Obj1`, and a length of N, you would type

```
v = kdt.Vec(N, element=kdt.Obj1())
```

where calling the `Obj1` constructor returns an instance with default values.

The details of the built-in objects are defined in `kdt/ObjMethods.py`. To read graph data in from files, you may need to customize this file to describe your data precisely [[**FIX**: clarify]].

4.1.2.3 Elemental operations

KDT's `Obj1` and `Obj2`object types provide a few fields that can be used as needed for your particular graph data. Because of this flexibility, the KDT developers cannot know what it means to (*e.g.*) add two edge- or vertex-objects in your particular context. The built-in object types come with a sample implementation of the basic operations, defined in the file `kdt/ObjMethods.py`. The approach taken is that edge-/vertex-objects will often have a numerical value (that can be added, ORed, etc., in the normal way) and one or more attribute values (that would not make sense to add, OR, etc., in the normal way). The sample operations typically perform the corresponding scalar operation on the `weight` element of the edge-/vector-object and either ignore or just copy the other elements, but you can customize this behavior for your specific needs.[1] Note that `ObjMethods.py` also includes the operations used for elemental multiplication and addition in the default implementation of path following; these may also need tailoring for your use of `Obj1` and `Obj2` objects.

For example, your use case might have edges whose values are start-time and end-time, with the duration implied by the difference between the two times. You could express that by using the `Obj2`'s `weight` and `category` fields, respectively, to denote start-time and end-time. You might want to define addition of two such edges as the time from the earlier start-time to the later end-time, which you could implement by modifying `__add__` method in `ObjMethods.py` as follows:

```
def __add__(self, other):
    ret = self.__copy__()
    if isinstance(other, (float, int, long)):
        ret.weight = self.weight + other
    elif isinstance(self, pcb.Obj2) and
            isinstance(other, pcb.Obj2):
        ret.weight = min(self.weight, other.weight)
        ret.category = max(self.category, other.category)
    return ret
```

Note that `__iadd__` and `__radd__` would need to be changed similarly. You'd also need to change the `_SR_mul_` definition in `ObjMethods.py` to the following:

```
def _SR_mul_(self, other):
    if isinstance(self, (float, int, long)):
        [ omitted for brevity … ]
    elif isinstance(self, pcb.Obj1):
        [ omitted for brevity … ]
    elif isinstance(self, pcb.Obj2):
        if isinstance(other, pcb.Obj2):
            self.weight = (self.category-self.weight)
                          * (other.category-other.weight)
        else:
            raise NotImplementedError
```

See Section 4.3.9 for details.

4.2 Algorithms

The KDT `DiGraph` class includes the algorithms in this section.

4.2.1 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

[1] For v0.2, note that `ObjMethods.py` resides in a KDT installation, so changing it changes its behavior for all users of that installation. If you want to tailor `ObjMethods.py` for your problem without effecting other KDT users on the same system, you may want to install KDT in a private location on the system.

```
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 `Vec` instance (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.

4.2.1.1 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 can be prohibitively expensive for large graphs, while approximate betweenness centrality is less computationally expensive.

Version 0.2

The performance of exact and approximate BC have not been fully characterized, but the following fragmentary results may be useful. The input graph was generated by `generate2DTorus`, whose diameter grows as the diameter of the torus. All times are in seconds.

N = # graph vertices `[twoDTorus(n)]`	N = 4096 (n = 64)			N = 16384 (n = 128)			N = 65536 (n = 256)
#cores	1	4	16	1	4	16	16
exactBC	229	75	33	5649	1970	1163	35010
approxBC (sample=0.05)	155	68		260		330	2717

4.2.1.2 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

See v0.2 timing results above in section 4.2.1.1. The numerical values of `'approxBC', sample=0.05` compared to `'exactBC'` are as follows for a graph generated by `generate2DTorus(256)`, which has equal centrality for all vertices. Exact result: 0.001938. Approximate result: mean 0.001944, standard deviation 0.000784.

4.2.1 PageRank

The PageRank algorithm [Page] computes a Web page's importance based on the number and importance of links to it.

Syntax

```
c = G.pageRank([epsilon = 0.1][, dampingFactor = 0.85])
```

Description

The `pageRank` function takes as input a `DiGraph` object and optional arguments and returns a `Vec` instance (of length equal to the number of vertices in the graph) that contains, for each respective vertex of the graph, the vertex's rank. The optional argument `epsilon` controls the stopping condition; iteration stops when the 1-norm of the difference in two successive result vectors is less than `epsilon`; the default is 0.1. The optional argument `dampingFactor` alters the results and speed of convergence, denoting the probability that the random surfer hops to an adjacent vertex (rather than hopping to a random vertex in the graph); the default value is 0.85.

The PageRank algorithm computes the percentage of time that a "random surfer" spends at each vertex in the graph. If the random surfer is at vertex V, she will take one of two actions:

- She will hop to another vertex to which vertex V has an outward edge (self loops are ignored).
- She will randomly hop to any vertex in the graph. This action is taken with probability (1 – dampingFactor)

4.2.2 Search

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

4.2.2.1 Breadth-first Search Tree

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

Syntax

```
parents = G.bfsTree(start)
```

Description

The `bfsTree` method takes as input a `DiGraph` instance and the index of the vertex from which to start the search. It returns a `Vec` (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.

Performance

The performance of the Graph500 benchmark is dominated by the performance of the `bfsTree` method. The fragmentary Graph500 performance available to date is described in section 1.4.

4.3 Methods

The KDT `DiGraph` and `Vec` classes implement the methods in this section.

4.3.1 Built-in methods for Vec vectors

Version 0.2

The `Vec` class represents dense and sparse vectors, which can be used to represent vertices and directed-edges. 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.

Syntax

In the following section, the variables `v`, `v2`, and `v3` are `Vec` instances, `k` is an integer scalar, and `m` is a scalar (integer or floating-point). For examples, these variables have the following values, with `v` and `v2` (and hence `v3`) assumed to be of length 11 and sparse:

v = [0, null, 4, null, 16, null, 36, null, 64, null, 100], v2 = [1, null, null, 27, null, null, 216, null, null, 729, null], and k = 4. Result values of v3 are denoted by "->".

```
v3 = v > k            # v3 is a Boolean vector, where each element
v3 = v >= k           #  is the comparison of the corresponding
v3 = v < k            #  element of v with the scalar k
v3 = v <= k           # v3 -> [1, null, 1, null, 0, null, 0, null, 0, null, 0, null]
v3 = v == k
v3 = v != k
v3 = v > v2           # v3 is a Boolean vector, where each element
v3 = v >= v2          #  is the comparison of the corresponding
v3 = v < v2           #  elements of v and v2
v3 = v <= v2
v3 = v == v2
v3 = v != v2
# bitwise operators following
v3 = ~v
v3 = v & v2
v3 = v | v2          # v3 -> [1, null, 4, 27, 16, null, 252, null, 64, 729, 100]
v3 = v ^ v2

v3 = v + k
v3 = k + v
v3 = v + v2
v += k
v += v2
v3 = -v
v3 = v – k
v3 = k - v
v3 = v – v2       # v3 -> [-1, null, 4, -27, 16, null, -180, null, 64, -729, 100]
v -= k
v -= v2
v3 = v * k
v3 = k * v
v3 = v * v2      # v3 -> [0, null, 0, 0, 0, null, 7776, null, 0, 0, 0]
v *= v2
v3 = v / k
v3 = k / v
v3 = v / v2
v3 = v % k
v3 = v % v2

v[k] = m
m = v[k]
```

```
v[vec] = m             # vec an integer vector  [[FIX: rm?]]
v2 = v[vec]                                       [[FIX: rm?]]
```

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

```
print v
v.printAll()
```

```
# following only apply to sparse Vec instances
del v[k]        # make the specified element null  [[FIX: rm?]]
del v[vec]      # make the specified elements null  [[FIX: rm?]]
```

Description

All elements of dense `Vec` instances are non-null (*i.e.,* have a value), while only explicitly created or assigned elements of sparse `Vec` instances are non-null. Operations only operate on non-null elements, which means all elements for dense `Vec` instances and (typically) not all elements for sparse `Vec` instances. For example, for `v2` a sparse `Vec` instance with a value of [1, null, null, 27, null, null, 216, null, null, 729, null], the result of `v2+4` would be [5, null, null, 31, null, null, 220, null, null, 733, null] and the (perhaps counterintuitive) result of `v2.min` would be 1.

Otherwise, these methods have their standard definitions in Python with the following exceptions:

- Only a key set of indexing ("`[]`") modes are supported. The right-hand side modes all return a `Vec` instance with length equal to the number of requested elements, unless otherwise noted.
  - Indexing on the right-hand side of an equation by a scalar, returning a scalar. This mode can be useful for debugging but will typically lead to poor performance if used otherwise.
  - Indexing on the right-hand side of an equation by a Boolean dense `Vec` instance. `[[FIX: rm?]]`
  - Indexing on the right-hand side of an equation by a non-Boolean dense `Vec` instance. `[[FIX: rm?]]`
  - Indexing on the right-hand side of an equation by a Boolean sparse `Vec` instance. `[[FIX: rm?]]`
  - Indexing on the right-hand side of an equation by a non-Boolean sparse `Vec` instance, returning a sparse `Vec` instance. `[[FIX: rm?]]`
  - Indexing on the left-hand side by a scalar index with a scalar value, changing a single value. This mode can be useful for debugging but will typically lead to poor performance if used otherwise.

  - Indexing on the left-hand side by a Boolean dense `Vec` index with a scalar value, changing the indicated values of the base dense `Vec` instance to the value of the scalar. `[[FIX: rm?]]`

  - Indexing on the left-hand side by a Boolean dense `Vec` index with a dense `Vec` value, changing the values of the base dense `Vec` instance corresponding to the True values of the dense `Vec` index to the corresponding values of the dense `Vec` value. `[[FIX: rm?]]`

  - Indexing on the left-hand side by a non-Boolean sparse `Vec` index vector with a scalar value. `[[FIX: rm?]]`

  - Indexing on the left-hand side by a Boolean sparse `Vec` index with a sparse `Vec` value, changing the indicated values of the base sparse `Vec` instance to the corresponding values from the value vector. `[[FIX: rm?]]`

- The modulus ("%") function implements the behavior of Python's modulus function for negative numbers, where the sign of the result is the same as the sign of the divisor (*e.g.*, "-1 % 8" yields "7"). This is different from the behavior of Python's `math.fmod` or C's `fmod`.

- The standard `print` statement will only print the first 100 (_REPR_MAX) elements of a `Vec` instance. Printing all the elements can be achieved via `printAll`.

4.3.2 Non-built-in methods for Vec vectors

The `Vec` class implements several other methods, some of which are common Python (or SciPy) names and some which are not.

The `Vec` class incorporates both dense and sparse vectors. Calls to `Vec` class constructors typically create a sparse vector by default, with the exception of the `ones`, `zeros`, and `range` methods, which are naturally dense. The default can be overridden by the `sparse` keyword, which accepts a Boolean value.

Syntax

```
# v and v2 are Vec instances as in the previous section
# k an integer scalar, m a floating-point scalar

v = Vec(length=0, element=0.0, sparse=True)
v = kdt.Vec.ones(size, element=1.0, sparse=False)
v = kdt.Vec.zeros(size, element=0.0, sparse=False)
v = kdt.Vec.range([start,] stop, element=0, sparse=False)

k = v.len()
k = v.nn()             # number of nulls;  k <- 5
k = v.nnn()            # number of non-nulls; k <- 6
k = v.nnz()            # number of nonzero nonnulls  [[FIX:  rm?]]
```

```
v2 = v.copy(element=None)  # deep copy
vSparse = v.sparse()       # convert a Vec to a sparse Vec
vDense = v.dense()         # convert a Vec to a dense Vec
boolScalar = v.isDense()
boolScalar = v.isSparse()
boolScalar = v.isObj()
boolScalar = v.isBool()
v.toBool()

v2 = v.abs()
v2 = v.ceil()         [[FIX:  rm?]]
v2 = v.floor()
v2 = v.round()        [[FIX:  rm?]]
v2 = v.sign()         [[FIX:  rm?]]

boolResult = v.any()  # boolResult is a Boolean scalar
boolResult = v.all()
boolResult = v.allCloseToInt()

v2 = v.logicalNot()   # also accessible via not keyword
v3 = v.logicalAnd(v2) # v3 <- [0, null, 0, null, 0, null, 1, null, 0, null, 0]
v3 = v.logicalOr(v2)
v3 = v.logicalXor(v2)

v3 = v.find(pred=None) # v.dense().find() -> [null, null, 4, null, 16, null, 36, null, 64, null, 100]
v3 = v.findInds(pred=None)   # v.dense().findInds() -> [2,4,6,8,10]

m = v.max(initInf=None)
m = v.min(initNegInf=None) # m -> 0
k = v.argmax(initInf=None)
k = v.argmin(initNegInf=None)  # for v=kdt.range(10), k <- 9
m = v.sum()
m = v.mean()
m = v.std()
m = v.norm(ord=k)     # k-norm, where k > 0

v.sort()
[sorted, perm] = v.sorted()

v3 = v.hist()         # v3 -> [15,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]

v.randPerm()          # only for dense

v = kdt.load(fName, element=0.0, sparse=True)
v.save(fName)
```

```
print v
v.printAll()

v.apply(op)           # applies the Python function op to every nonnull
v.applyInd(op)
m = v.reduce(op, uniOp=None, init=None)
k = v.count(op, pred=None)

v.set(value)

# functions that work only for sparse Vec instances
v.spOnes
v.spRange()

v.addFilter(filter)
v.delFilter(filter=None)
```

Description

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

The `Vec` constructor creates a `Vec` instance of the length, element-type, and sparsity (density) specified. Note that length denotes the maximum number of positions in the vector; for a sparse vector, all of these positions are initially null. The `ones` and `zeros` methods creates a `Vec` instance of the length specified with all elements set to 1 and 0, respectively. The `range` method creates a `Vec` 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 `nn` and `nnn` methods return the number of nulls and nonnulls, respectively, with `nnn` behaving the same as SciPy's `getnnz`. The `nnz` method returns the number of nonnulls that are not equal to zero. The `findInds` method returns the indices of nonnull elements of the input `Vec` instance, the same as NumPy/SciPy's `nonzero` method. The distinct names were chosen to provide future flexibility for a sparse `Vec` class with a configurable null (zero) element.

Assignment ("=") of a `Vec` 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 `element` argument, if specified, can coerce a `Vec` instance whose element is an `Obj1` or `Obj2` instance to a `Vec` instance whose element is a 64-bit element. The `sparse` method converts a dense `Vec` to a sparse `Vec` , or returns the input if already sparse. Similarly, the `dense` method converts a sparse `Vec` to a dense `Vec` , or returns the input if already dense. The `iDense` and `isSparse` methods return True if the input vector is dense or sparse, respectively. The `isObj` method returns True if the elemental value of the `Vec` instance is either an `Obj1` or `Obj2` instance and False if it's the default 64-bit element. The `isBool` method returns True if all elements of the vector are True (1) or False (0). The `toBool` method converts each nonnull element of the vector in place to False (0) if its value is zero and True (1) otherwise.

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

The `logicalNot` method returns the logical negation of each (nonnull) element of the input vector. The `logicalAnd`, `logicalOr`, and `logicalXor` methods return the element-wise logical And, Or, and Xor, respectively, of the two input vectors. By contrast, for input values 0x55 and 0xAA, the built-in bitwise & operator will return 0x00 (no corresonding bit is set in both values), while the `logicalAnd` method will return True (at least one bit is set in both values).

The `find` method returns the nonzero elements of a dense `Vec` instance, in a sparse `Vec` instance. The `findInds` method returns the indices of the nonzero elements of a dense `Vec` instance, in a dense `Vec` instance.

The `max` and `min` methods calculate the maximum and minimum value, respectively, in the vector. Note that for sparse `Vec` instances, even though null values are often viewed as having zero value, the `min` and `max` methods, like other KDT `Vec` methods, operate just on the non-null elements, and thus (e.g.) taking the minimum of the vector `v2` returns 1, even though null elements exist. With the optional `initInf` and `initNegInf` arguments, you can specify the initial value for positive and negative infinity, respectively. **Note:** for `v` a `Vec` instance, the supported interface is `v.min()` ; `min(v)` sometimes gives a different answer, and similarly for `max`.

The `argmax` and `argmin` methods return the index of the maximum and minimum non-null element, respectively. Currently they are only implemented for dense `Vec` instances. The `mean` and `std` methods calculate the mean and standard deviation, respectively, of the (nonnull values of the) input vector. The `norm` method (modeled on the NumPy method) calculates the k-norm of the vector, where k is greater than zero.

The `sort` method sorts the elements of the input vectorin ascending order and returns, in-place, the permutation that creates that order. The `sorted` method returns a tuple whose first element is a copy of the input vector, sorted in ascending order, and whose second element is the indices of the sorted elements in the original vector. If the input vector is sparse, the output vectors will be as well. **Note:** the `sort` and `sorted` methods do not support input `Vec` instances that have filters.

The `hist` method returns the histogram of the input in a dense `Vec` instance of length equal to the lesser of the maximum value in the input vector rounded to an integer or the length of the input vector.

The `randPerm` method randomly permutes the elements of the `Vec` instance, which must be dense.

The `load` and `save` methods load and save, respectively, a `Vec` instance from (to) a file of the given name. The file must be in the Coordinate Format of the Matrix Market Exchange Formats (see also Section 4.6.1). Currently, the result `Vec` instance from the `load` method will be created with an elemental 64-bit value; in the future, this may be overridden with the `element` argument specifying either an `Obj1` or `Obj2` instance. Similarly, the input `Vec` instance from the `load` method must have an elemental 64-bit value. Also by default the `Vec` instance will be created sparse; this can be overridden with the `sparse` argument.

The `apply`, `applyInd`, `reduce`, and `count` methods are highly configurable via their `op` (and `uniOp`) arguments, which are Python operations that define the action to be taken for each element of the `Vec` instance. See Section 4.3.11 for more details. The `apply` method applies the `op` function to each (nonnull, in the case of a sparse vector) element of the `Vec` instance, changing its value in place. The `applyInd` method applies the `op` function to the index of each (nonnull, in the case of a sparse vector) element of the `Vec` instance, changing its value in place. The `reduce` method applies the `uniOp` function to each (nonnull, in the case of a sparse vector) element of the `Vec` instance and then applies the binary `op` function to that result and the current reduction value; the result of the binary function will be a single element (which can be a 64-bit element or an `Obj1` or `Obj2` instance). The `count` method applies the `pred` function to each (nonnull, in the case of a sparse vector) element of the `Vec` instance; if the Boolean return value is True, it will then apply the binary `op` function to that result and the current reduction value; the result of the binary function will be a single element (which can be a 64-bit element or an `Obj1` or `Obj2` instance).

The `set` method sets every (nonnull, in the case of a sparse vector) element of the `Vec` instance to `value`, which must be of the same type (64-bit or `Obj1` or `Obj2` instance) as the element of the `Vec` instance.

The `spOnes` method modifies the bound sparse `Vec` instance by setting all its nonnull elements to 1.0. The `set` method modifies the bound `Vec` instance by setting all its nonnull elements to the passed value. The `spRange` method modifies the bound sparse `Vec` instance by setting each of its nonnull elements to its index in the vector.

The `addFilter` method adds the Python function denoted by the `filter` argument to the `Vec` instance, with semantics as described in Section 4.1.2. The `delFilter` removes the previously-attached filter indicated by the `filter` argument; a null `filter` argument indicates to remove all filters from the `Vec` instance.

4.3.3 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 0.2

`DiGraph` instances support a 64-bit floating-point, an `Obj1` instance, or an `Obj2` instance as elemental datatype and a Boolean datatype, with an explicit conversion required to the Boolean form.

Syntax

```
# G1, G2 and G3 are DiGraphs
G3 = G1 + G2          # elemental addition
G3 += G2              # elemental addition in place
G3 = -G1              # elemental negation
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 `toBool` method converts each nonnull element of a `DiGraph` instance to a Boolean True, thereby consuming less space and enabling faster operations.

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 v0.2, 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 via `Vec.printAll`.

4.3.4 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.

4.3.4.1 Simple methods

Syntax

```
# G and G2 are DiGraphs
G = kdt.DiGraph(sourceV=None, destV=None, valueV=None, nv=None,
      element=0, edges=None, vertices=None)
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.toBool()             # set all non-null edge-weights to True
G.set(k)               # set all non-null edge-weights to k
G.mulNot(G2)           # see below
G.removeSelfLoops()    # remove all edges from a vertex to itself


G.addFilter(filter)
G.delFilter(filter=None)
```

Description

These methods have their obvious definitions with the following elaborations.

The `DiGraph` constructor creates a `DiGraph` instance and is typically used in one of the following ways:

- By passing in vectors representing the source vertex, destination vertex, and value for each edge in the directed graph, along with the number of vertices (`nv`). Note that the `valueV` argument can have an elemental type of a 64-bit double, an `Obj1` instance, or an `Obj2` instance.
- Like above, but overriding the elemental type with the `element` argument.
- By passing in a previously created `Mat` class instance representing the edges, with the `vertices` argument. See Sections 4.3.6 and 4.3.7 for more details.
- Like any of the above, but also providing a vector of vertex attributes with the `vertices` argument.

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.

The `toBool` method converts each nonnull element of a `DiGraph` instance to a Boolean True, thereby consuming less space and enabling faster operations.

The `mulNot` method 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.

The `addFilter` method adds the Python function denoted by the `filter` argument to the `DiGraph` instance, with semantics as described in Section 4.1.2. The `delFilter` removes the previously-attached filter indicated by the `filter` argument; a null `filter` argument indicates to remove all filters from the `DiGraph` instance.

4.3.4.2 DiGraph

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

Syntax

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

Description

The `DiGraph` method creates a `DiGraph` instance. The required input parameters `source` and `dest` are `Vec` instances created by the program or by generators such as `generateRMAT`. The required input parameter `weight` can be a `Vec` instance 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 `generateRMAT`, 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 = kdt.Vec.zeros(N)
dest = kdt.Vec.range(N)
G = kdt.DiGraph(source, dest, 1, N)
```

4.3.4.3 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 `Vec` 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.

4.3.4.4 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 overwrites its input contents.

4.3.4.5 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.subgraph(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 vertex indices) of vertices (and out-edges incident to them) to be used for the new `DiGraph` instance. The optional input argument `vertrange2` specifies another range of vertices (and in-edges incident to them) to be used as the in-vertices 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(kdt.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(kdt.DiGraph.range(nvG/2), kdt.DiGraph.range(nvG/2,
nvG))
```

4.3.4.6 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=kdt.DiGraph.Out])
inoutsums = G.sum([dir=kdt.DiGraph.Out])
inoutmaxs = G.max([dir=kdt.DiGraph.Out])
inoutmins = G.min([dir=kdt.DiGraph.Out])
```

Description

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

4.3.4.7 scale

[[**FIX**: ensure DiGraph version exists.] 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=kdt.DiGraph.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 sparse `Vec` 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 = kdt.Vec.ones(G.nvert()) / G.sum()
G.scale(scalefac)
```

4.3.4.8 normalizeEdgeWeights

The `normalizeEdgeWeights` method of the `DiGraph` class normalizes the out-edge weights of each vertex V of a `DiGraph` instance such that each out-edge's weight is 1 / out-degree(V).

Syntax

```
G.normalizeEdgeWeights()
```

Description

The `normalizeEdgeWeights` method of the `DiGraph` class normalizes the out-edge weights of each vertex V of a `DiGraph` instance such that each out-edge's weight is 1 / out-degree(V). Besides the `DiGraph` instance there are no other arguments.

4.3.5 Advanced methods for DiGraph objects

The `DiGraph` class implements several advanced methods.

4.3.5.1 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)
```

Description

The `neighbor` method of the `DiGraph` class calculates, for a `Vec` instance containing 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 return value is a `Vec` with the indices of neighboring vertices.

Example

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

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

4.3.5.2 Breadth-first Search Tree Validation

The `isBfsTree` method verifies that a breadth-first search tree of a `DiGraph` instance is valid.

Syntax

```
[valid, levels] = G.isBfsTree(start, parents)
```

Description

The `isBfsTree` method takes as input a `DiGraph` instance, the index of the vertex from which the BFS-tree construction started, and the parent of each vertex (as returned by `bfsTree`). It returns a 2-tuple whose first element is 1 if the BFS tree is valid and the negative of the number of the test below that failed if the tree is invalid. The second element of the tuple is a `Vec` instance (of length equal to the number of vertices in the graph) that denotes, for each respective vertex of the graph, the vertex's level in the BFS tree. Just as for `bfsTree`, the `start` vertex is its own parent, and a vertex that is unreachable from the `start` vertex has a parent of -1.

The validity tests include:

- [1] The tree does not contain cycles, every vertex with a parent is in the tree, and the root is not the destination of any tree edge.
- [2] Tree edges are between vertices whose levels differ by exactly 1.

4.3.6 [For algorithm developers] Built-in methods for Mat objects

Besides KDT's primary audience of domain experts who need to do graph analysis, another important audience is graph-algorithm developers wanting to implement and test their algorithms on very large data. For these algorithm developers, KDT's implementation of directed graphs and hypergraphs on underlying distributed sparse matrices is exposed, with the `Mat` class available for coding algorithms based on linear algebra. This section and the next describe the methods implemented for `Mat` objects, some of which are mapped onto standard Python operators or methods consistent with clear understandability.

Version 0.2

`Mat` instances support a 64-bit floating-point, an `Obj1` instance, or an `Obj2` instance as elemental datatype and a Boolean datatype, with an explicit conversion required to the Boolean form.

Syntax

```
# M1, M2 and M3 are Mats
# v1, v2 are Vecs
```

```
M3 = M1 + M2          # elemental addition
M3 += M2              # elemental addition in place
M3 = M1 * M2          # elemental multiplication
M3 *= M2              # elemental multiplication in place
M3 = M1 / M2          # elemental division
M3 = -M1              # elemental negation

M2 = M[v]             # FIX:  which subsref cases handled?
M2 = M[v1, v2]

print M
```

Description

For each of these elemental arithmetic operations, the two matrices must have the same number of rows and the same number of columns. These methods have their obvious definitions, with the caveat that the multiplication and division operators yield a null value in any element for which at least one of the graphs has a null entry. Because `Mat` instances are often large objects, in-place operators may often make sense to conserve the amount of memory consumed by temporary or transient objects.

Note the `mulNot` method in the next section; it multiplies one matrix by the logical negation of a second.

The `print` method for `Mat` objects can be problematic, as they can often be extremely large (billions of elements), for which text display is rarely useful. For v0.2, for small graphs (defined as 100 edges or fewer), the print method will call the `toVec` method and print those results.

4.3.7 [For algorithm developers] General-purpose methods for Mat objects

The following methods are defined for `Mat` instances.

```
# M and M2 are Mats
# k is an integer scalar, m is a floating-point scalar

M = kdt.Mat(i=None, j=None, v=None, n=None, m=None, element=0)

M = kdt.Mat.load(fname)
M.save(fname)
[M, degV, time] = kdt.Mat.generateRMAT(scale, edgeFactor=16,
       initiator[0.57, 0.19, 0.19, 0.05], delIsolated=True,
       element=True)
M = kdt.Mat.eye(n, m=None, element=1.0)
M = kdt.Mat.ones(n, m=None, element=1.0)

k = M.nrow()          # number of rows
k = M.ncol()          # number of columns
k = M.nnn()           # number of non-null elements
boolScalar = M.isScalar()  # is M’s element a 64-bit floating-point?
```

```
boolScalar = M.isBool()    # is M's element a Boolean?
boolScalar = M.isObj()     # is M's element an Obj1 or Obj2 instance?

M2 = M.copy(element=)      # deep copy
M2 = M.toBool(inPlace=True)
M2 = M.toScalar(inPlace=True)
M.spOnes()
M.removeMainDiagonal()
M.transpose()

[i, j, v] = M.toVec()

M.mulNot(M2)          # see below

m = M.sum(dir=kdt.Mat.Column)
m = M.max(dir=kdt.Mat.Column)
m = M.min(dir=kdt.Mat.Column)
m = M.reduce(dir, op, unOp=None, init=None)

M.scale(otherV, op=op_mul, dir=kdt.Mat.Column)

v = M.SpMV(other, semiring, inPlace=False)
M.SpMV(other, semiring, inPlace=True)
M2 = M.SpGEMM(other, semiring, inPlace=False)
M.SpGEMM(other, semiring, inPlace=True)

M.apply(op, other=None, notB=False)

M2 = M.eWiseApply(other, op, allowANulls=False, allowBNulls=False,
         doOp=None, inPlace=False)
M.eWiseApply(other, op, allowANulls=False, allowBNulls=False,
         doOp=None, inPlace=True)

M.addFilter(filter)
M.delFilter(filter=None)
```

Description

These methods have their usual definitions with the following elaborations.

The `Mat` constructor creates a `Mat` instance and is typically used in one of the following ways:

- By passing in vectors representing the column index, row index, and elemental value for each non-null element in the sparse matrix, along with the number of columns (and optionally number of rows, if it's different from the number of columns). Note that the `v` argument can have an elemental type of a 64-bit double, an `Obj1` instance, or an `Obj2` instance.

- Like above, but overriding the elemental type with the `element` argument.

The `load` constructor (and its accompanying `save` method) load (save) a `Mat` instance from (to) a file of the given name. The file must be in the Coordinate Format of the Matrix Market Exchange Formats (see also Section 4.6.1). **Note that the result `Mat` instance will be the transpose of the MatrixMarket definition.** By default, the `Mat` instance will be created with an elemental 64-bit value; this can be overridden with the `element` argument specifying either an `Obj1` or `Obj2` instance.

The `generateRMAT` constructor accepts the `scale` argument, which is the integer logarithm (base 2) of the number of rows and columns in the resulting graph, the `fillFactor` argument, which is the average number of nonzeros in each row and column, the `initiator` argument, which is a 4-tuple denoting the probability of a new non-null occurring in each quadrant of the (sub)matrix, with default values of [0.57, 0.19, 0.19, 0.05] for upper-left, upper-right, lower-left, and lower-right, respectively, the `delIsolated` argument, which if True will delete a diagonal element whose corresponding rows and columns have no nonnull entries, and the `element` argument, which denotes the type of the elemental value, with a default of Boolean. The `generateRMAT` constructor returns the created MAT instance, a `Vec` instance denoting the number of non-nulls in each column of the matrix, and a floating-point scalar denoting the time required to create the matrix.

The `eye` constructor creates an identity matrix of the number of columns (and optionally number of rows; only needed if it's different from the number of columns) indicated. The `ones` constructor creates a `Mat` instance with every element of every row and column set to 1.0 by default, or the value of the `element` argument if specified.

The `nrow` and `ncol` methods, when called on a `Mat` instance, return the number of rows and columns, respectively. The `nnn` method returns the number of non-null elements in the `Mat` instance. The `isScalar` method returns a scalar Boolean result that is True if the elemental value of the `Mat` instance is a 64-bit floating point. The `isBool` method returns a scalar Boolean result that is True if the elemental value of the `Mat` instance is a Boolean. The `isObj` method returns a scalar Boolean result that is True if the elemental value of the `Mat` instance is an `Obj1` or `Obj2` instance.

Assignment ("=") of a `Mat` 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 `element` argument dictates the type of the elemental value of the resulting `Mat` instance. The `toBool` method returns a `Mat` object whose elemental value of the `Mat` instance is a Boolean and whose elements are True in exactly those positions where the original matrix was nonzero; the optional `inPlace` argument will cause the original matrix to be overwritten. The `toScalar` method returns a `Mat` object whose elemental value of the `Mat` instance is a 64-bit floating-point value and whose elements are nonzero in exactly those positions where the original matrix was nonzero; the optional `inPlace` argument will cause the original matrix to be overwritten. The `spOnes` method returns a `Mat` object whose elements are equal to 1.0 in exactly those positions where the original matrix was nonzero. The `removeMainDiagonal` method removes, in place, all non-null elements of the bound `Mat` instance. The `transpose` method tranposes, in place, the bound `Mat` instance.

The toVec method returns the column, row, and value of each nonnull element of a Mat instance in the i, j, and v Vec instances, respectively.

The mulNot method takes the logical inverse of each element of the second Mat instance before element-wise multiplying the respective elements of the two Mat instances. Because it does this elementally, there is no extra memory consumed by the inverse values.

The sum, max, and min methods calculate the sum, maximum, and minimum, respectively of the rows or columns of the Mat instance. By default the reduction is calculated within the columns of the matrix; this can be overwritten by the dir argument. The reduce method is a generalized reduction, accepting the same dir argument, and also the op argument, which specifies the binary reduction to be performed by the reduction, and the unOp argument, which specifies a unary operator to be applied to the elemental value before the reduction is performed. The optional init argument can set the initial value of the reduction to something other than its default of 0 [[**FIX**: correct?]].

The scale method modifies each element of a column (row) of a Mat instance by the corresponding element of a Vec instance (which must have length equal to the number of columns (rows). The default scaling is to multiply by the scale factor; this behavior can be overridden by specifying the op argument, which must be a Python function accepting two arguments and returning a single argument.

The SpMV method, by default, performs a standard sparse-matrix times vector multiplication of a Mat instance and a Vec instance. The operation by default returns a new Vec instance; specifying the inPlace argument will modify the input Vec argument instead, with no return argument. By specifying the semiring argument, the default behavior of the standard elemental multiplication and sum reduction can be overridden. A semiring is composed of an ‘add’ function and a ‘multiply’ function, joined together by the kdt.Util.sr method. [[**FIX**: describe somewhere in more detail.]]

The SpGEMM method, by default, performs a standard sparse-matrix times sparse-matrix multiplication of two Mat instances. The operation by default returns a new Mat instance; specifying the inPlace argument will modify the first input Mat argument instead, with no return argument. By specifying the semiring argument, the default behavior of the standard elemental multiplication and sum reduction can be overridden. A semiring is composed of an ‘add’ function and a ‘multiply’ function, joined together by the kdt.Util.sr method.

The apply method applies a unary Python function specified by the op argument to each element of the Mat instance or, if the optional other argument is supplied, a binary function can be provided. The value of the other argument can optionally be negated before the operation is performed by specifying the notB argument with a value of Boolean True.

The eWiseApply method applies a binary Python function specified by the op argument to corresponding elements of two Mat instances. By default the operation will be applied only where both Mat instances have non-null elements; the allowANulls argument with a value of True will cause it to be applied wherever the second Mat instance is non-null and the allowBNulls argument with a value of True will cause it to be applied wherever the first `Mat` instance is non-null. Specifying both the `allowANulls` and `allowBNulls` arguments as True will cause it to be applied wherever either `Mat` instance is non-null. As elsewhere, the `inPlace` argument controls whether the first argument is overwritten by the new data; the default is to return a new `Mat` instance.

The `addFilter` method adds the Python function denoted by the `filter` argument to the `Mat` instance, with semantics as described in Section 4.1.2. The `delFilter` removes the previously-attached filter indicated by the `filter` argument; a null `filter` argument indicates to remove all filters from the `Mat` instance.

4.3.8 Miscellaneous methods

The following methods exist in the KDT package but not in any class.

- `version`: returns the KDT version number as a string.
- `revision`: returns the SVN revision number (of the KDT release contents) as a string.
- `sendFeedback`: as described in section 1.6, sends feedback to the KDT development team.
- `master`: as described in section 4.6.4, can be used to restrict output to the master process.

4.3.9 Semantic graph details

4.3.9.1 Semantic graphs via `DiGraph` and `Vec` constructors

Semantic graph data will typically be coming from the real world, and so will be loaded from files. The `DiGraph.load` and `Vec.load` methods depend on the `load` function in `ObjMethods.py` to understand the layout of the file data. Semantic graphs can also be built from a vector of edge-object elements by using the `element` argument to the `DiGraph` constructor.

The `DiGraph.copy` (`Vec.copy`) methods accept an `element` argument that can be used to convert a `DiGraph` (`Vec`) instance with `Obj1` or `Obj2` elements to a `DiGraph` (`Vec`) instance with the non-semantic 64-bit element.

4.3.9.2 addEFilter

Syntax

```
G.addEFilter(filter)
```

Description

The `addEFilter` method attaches the passed edge-filter to the given `DiGraph` instance. The filter must be a `PyObject` object that accepts as input the elemental object of the `DiGraph` instance (`Obj1` or `Obj2`) and returns a Boolean value. The filter will be invoked internally to all subsequent methods on the `DiGraph` instance until the filter is deleted via `delEFilter` or the `DiGraph` instance is deleted or goes out of scope. Multiple filters may be attached; they can be viewed as executing in series (for each edge) until all have returned True or until a filter returns False. Only edges for which the filter(s) returns a Boolean True (and which are not incident to vertices which do not pass a filter) will be used for the remainder of the method. This method executes in-place and does not provide a return value.

4.3.9.3 delEFilter

Syntax

`G.delEFilter([filter])`

Description

The `delEFilter` method deletes edge-filter(s) from the given `DiGraph` instance. If the optional argument is passed, it must be a `PyObject` object previously passed to `addEFilter` on the same `DiGraph` instance. If the optional argument is not passed, all filters are deleted from the `DiGraph` instance. This method executes in-place and does not provide a return value.

4.3.9.4 addVFilter

Syntax

`v.addVFilter(filter)`

Description

The `addVFilter` method attaches the passed vertex-filter to the given `Vec` instance (which may be the vertex attributes of a `DiGraph` instance (see section 4.3.9.6)). The filter must be a `PyObject` object that accepts as input the elemental object of the `Vec` instance (`Obj1` or `Obj2`) and returns a Boolean value. The filter will be invoked internally to all subsequent methods on the `Vec` instance (and any `DiGraph` instance to which the `Vec` instance is attached as a vertex filter) until the filter is deleted via `delVFilter` or the `Vec` instance is deleted or goes out of scope. Multiple filters may be attached; they can be viewed as executing in series (for each vertex) until all have returned True or until a filter returns False. Only vertices for which the filter(s) returns a Boolean True will be used for the remainder of the method; vertices for which the filter(s) returns a Boolean False will have their incident edges viewed as not having passed the filter as well and thereby ignored for the remainder of the method. This method executes in-place and does not provide a return value.

4.3.9.5 delVFilter

Syntax

`v.delVFilter([filter])`

Description

The `delVFilter` method deletes vertex-filter(s) from the given `DiGraph` instance (which may be the vertex attributes of a `DiGraph` instance (see section 4.3.9.6)). If the optional argument is passed, it must be a `PyObject` object previously passed to `addVFilter` on the same `Vec` instance. If the optional argument is not passed, all filters are deleted from the `Vec` instance. This method executes in-place and does not provide a return value.

4.3.9.6 Adding or modifying vertex attributes to a `DiGraph` instance

Attributes of the vertices of a `DiGraph` instance can be specified by setting the `vAttr` attribute of the `DiGraph` instance to a `Vec` instance whose element is the desired object-type and data.

Syntax

```
G.vAttr = v
```

```
G.vAttr.addVFilter(filter)
G.vAttr.delVFilter([filter])
```

Description

The `vAttr` attribute will be queried for vertex filters whenever operations are performed on the `DiGraph` instance. Only vertices for which the filter(s) returns a Boolean True will be used for the remainder of the method; vertices for which the filter(s) returns a Boolean False will have their incident edges viewed as not having passed the filter as well and thereby ignored for the remainder of the method.

A `Vec` instance that is the `vAttr` attribute of a `DiGraph` instance may still be modified by adding or removing filters, as shown by the syntax above.

4.3.9.7 Semantic-graph example

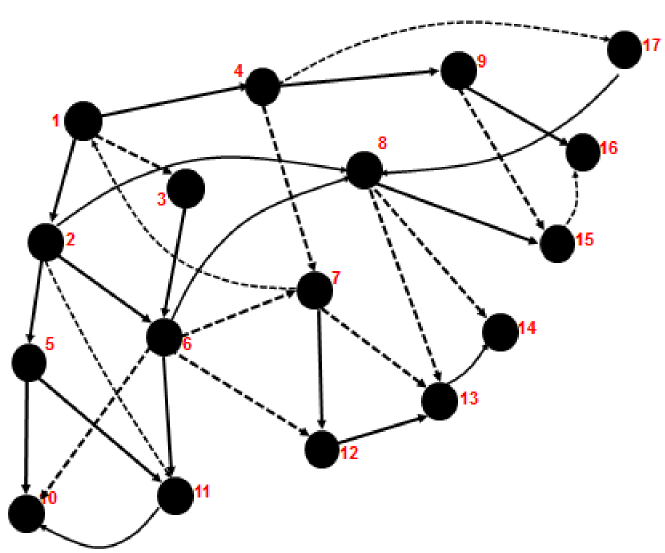

Figure 1. Sample semantic graph with two types of edges

A simple synthetic example of the use of filters is given below. The code is from the `test/TestDiGraph.py` file of the KDT distribution. It builds a semantic `DiGraph` instance of the graph in Figure 1 with `Obj1` edge-elements split between categories 1 and 2 and calculates the degree of each vertex looking first at just category 1 and then category 2 edges.

```
def test_degree_Out_Obj1_filteredTwoWays(self):
    # filters that test if category is equal to 1 or 2, respectively
    def category_eq_1(x):
            if isinstance(x, (Obj1, Obj2)):
                    return x.category==1
            else:
                    raise NotImplementedError
    def category_eq_2(x):
            if isinstance(x, (Obj1, Obj2)):
                    return x.category==2
            else:
                    raise NotImplementedError
    nvert = 18
    nedge = 31
    i = [7, 1, 1, 1, 2, 2, 3, 4, 6, 2, 6,17, 4, 5, 6,11, 2, 5, 6, 6,
7, 7, 8,12, 8,13, 8, 9, 9,15, 4]
    j = [1, 2, 3, 4, 5, 6, 6, 7, 7, 8, 8, 8, 9,10,10,10,11,11,11,12,
12,13,13,13,14,14,15,15,16,16,17]
    w = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    c = [2, 1, 2, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 2,
1, 2, 2, 1, 2, 1, 1, 2, 1, 2, 2]
    self.assertEqual(len(i), nedge)
    element = Obj1()
    G = self.initializeGraph(nvert, nedge, i, j, (w, c),
            element=element)
    G.addEFilter(category_eq_1)
    degOutCat1 = G.degree(DiGraph.Out)
    G.delEFilter(category_eq_1)
    expectedOutDegCat1 = [ 0, 2, 3, 1, 1, 2, 2, 1, 1, 1, 0, 1, 1, 1,
0, 0, 0, 1]
    self.assertEqual(len(degOutCat1), nvert)
    for ind in range(G.nvert()):
            self.assertEqual(expectedOutDegCat1[ind], degOutCat1[ind])
    G.addEFilter(category_eq_2)
    degOutCat2 = G.degree(DiGraph.Out)
    G.delEFilter(category_eq_2)
    expectedOutDegCat2 = [ 0, 1, 1, 0, 2, 0, 3, 2, 2, 1, 0, 0, 0, 0,
0, 1, 0, 0]
    self.assertEqual(len(degOutCat2), nvert)
```

```
for ind in range(G.nvert()):
        self.assertEqual(expectedOutDegCat2[ind], degOutCat2[ind])
```

4.3.9.8 Performance

Calling Python functions to filter and calculate element-wise is substantially slower than calling the built-in C++ functions used for non-semantic graphs. [[FIX: any data?]]

4.3.10 [For expert users] Details of semi-rings

[[**FIX**: fill in]

4.3.11 [For expert users] Details of `apply`, `applyInd`, and `reduce` methods

The `apply`, `applyInd`, `reduce`, and `count` methods are highly configurable via their `op` (and `uniOp`) arguments, which are Python operations that define the action to be taken for each element of the `Vec` instance. These methods may be used to provide functionality that is specific to your application or that is not implemented in KDT.

4.3.11.1 The `apply` method

The `apply` method accepts as its `op` argument a unary Python function whose input and output are an element of a `Vec` or `Mat` instance. The function will be called for each (non-null) element of the `Vec` or `Mat` instance, and the result of the function will be the new value of that element. The function can call any Python function that can be supported within the KDT infrastructure. [[**FIX**: need to characterize those that can (not) be called.]] The following example replaces each element of a `Vec` or `Mat` instance by its cosine, assuming that a 64-bit elemental value.

```
import kdt

def cos(x):
        import math
        return math.cos(x)

sz = 25
vec = kdt.Vec(sz)
print 'isSparse(vec)=%i' % vec.isSparse()
i = [0, 2, 4, 6, 8, 10]
v = [1, 4, 8,12,16, 20]
for ndx in range(len(i)):
        vec[i[ndx]] = v[ndx]
print vec
vec.apply(cos)
print vec
```

The output from running this script in IPython is as follows.

```
In [20]: %run try1
isSparse(vec)=1
Elements stored on proc 0: {(0,1), (2,4), (4,8), (6,12), (8,16),
(10,20), }
```

```
Elements stored on proc 0: {(0,0.540302), (2,-0.653644), (4,-0.1455),
(6,0.843854), (8,-0.957659), (10,0.408082), }


In [21]:
```

Extending this example so the elemental function will work with either a 64-bit, Obj1 instance or Obj2 instance results in code that looks like the following.

```
import kdt

def cos(x):
        import math
        if isinstance(x, (float, int, long)):
                x = math.cos(x)
        else:
                x.weight = math.cos(x.weight)
        return x

sz = 25
element = kdt.Obj1()
vec = kdt.Vec(sz, element=element)
print 'isSparse(vec)=%i' % vec.isSparse()
i = [0, 2, 4, 6, 8, 10]
v = [1, 4, 8,12,16, 20]
for ndx in range(len(i)):
        if isinstance(element, (float, int, long)):
                vec[i[ndx]] = v[ndx]
        else:
                element.weight = v[ndx]
                vec[i[ndx]] = element
print vec
vec.apply(cos)
print vec
```

The IPython output of this version looks like the following.

```
In [19]: %run try2
isSparse(vec)=1
Elements stored on proc 0: {(0,[ 1.000000, 0 ]), (2,[ 4.000000, 0 ]),
(4,[ 8.000000, 0 ]), (6,[ 12.000000, 0 ]), (8,[ 16.000000, 0 ]), (10,[
20.000000, 0 ]), }

Elements stored on proc 0: {(0,[ 0.540302, 0 ]), (2,[ -0.653644, 0 ]),
(4,[ -0.145500, 0 ]), (6,[ 0.843854, 0 ]), (8,[ -0.957659, 0 ]), (10,[
0.408082, 0 ]), }


In [20]:
```

4.3.11.2 The `applyInd` method

Similarly, the `applyInd` method applies a function to every element of a `Vec` instance, but the elemental function has two input values: the elemental value and the index of the elemental value, in the range of 0 to `len(vec)-1` inclusive. The following code illustrates this by setting the value of the element to the proportion of its position in the vector (in the range of 0.0 to 1.0 inclusive).

```
import kdt

sz = 25
def proportion(elem, elemNdx):
        if isinstance(elem, (float, int, long)):
                elem = float(elemNdx) / float(sz)
        else:
                elem.weight = float(elemNdx) / float(sz)
        return elem

element = 0
vec = kdt.Vec(sz, element=element)
print 'isSparse(vec)=%i' % vec.isSparse()
i = [0, 2, 4, 6, 8, 10]
v = [1, 4, 8,12,16, 20]
for ndx in range(len(i)):
        if isinstance(element, (float, int, long)):
                vec[i[ndx]] = v[ndx]
        else:
                element.weight = v[ndx]
                vec[i[ndx]] = element
                #print 'i[ndx]=%i, v[ndx]=%d, vec[i[ndx]].weight=%d' %
(i[ndx], v[ndx], vec[i[ndx]].weight)
print vec
vec.applyInd(proportion)
print vec
```

The output of this function from IPython follows.

```
In [36]: %run try3
isSparse(vec)=1
Elements stored on proc 0: {(0,1), (2,4), (4,8), (6,12), (8,16),
(10,20), }

Elements stored on proc 0: {(0,0), (2,0.08), (4,0.16), (6,0.24),
(8,0.32), (10,0.4), }

In [37]:
```

Note also that this example takes advantage of the the Python semantics for function definition by defining the value of the variable `sz` before the function definition, then using the value of `sz` (at the time of function definition) inside the function without passing it as an argument.

4.3.11.3 The `reduce` method

The `reduce` method combines elemental values in a user-defined way, either from a whole `Vec` instance or across one dimension of a `Mat` instance, reducing the dimensionality by one. The reduction is performed by a binary Python function passed via the `op` argument.

For example, we might have a vector whose elements are pairs of non-negative integers denoting a Manhattan distance from an origin. These could be represented by `Obj1` instances, using the `weight` and `category` fields. We might define the “distance” of an element as the sum of the x and y distances and want to calculate the maximum value over such a vector. The code might look like the following.

```
import kdt

sz = 25
def max2d(currentVal, elem):
        if isinstance(elem, kdt.Obj1):
                currentVal.weight = max(currentVal.weight,
                                        elem.weight+elem.category)
        return currentVal

element = kdt.Obj1()
element.weight = -1
element.category = -1
vec = kdt.Vec(sz, element=element)
print 'isSparse(vec)=%i' % vec.isSparse()
i = [0, 2, 4, 6, 8, 10]
v1 = [1, 4, 8,12,16, 20]
v2 = [8, 5, 3, 2, 1, 1]
for ndx in range(len(i)):
        if isinstance(element, (float, int, long)):
                vec[i[ndx]] = v[ndx]
        else:
                element.weight = v1[ndx]
                element.category = v2[ndx]
                vec[i[ndx]] = element
print vec
m = vec.reduce(max2d)
print m
```

The resulting IPython output is as follows.

```
In [60]: %run try4
isSparse(vec)=1
Elements stored on proc 0: {(0,[ 1.000000, 8 ]), (2,[ 4.000000, 5 ]),
(4,[ 8.000000, 3 ]), (6,[ 12.000000, 2 ]), (8,[ 16.000000, 1 ]), (10,[
20.000000, 1 ]), }

[ 21.000000, 8 ]
In [61]:
```

Obviously a better user interface for this example's functionality would be to override the elemental function called by the `max` method, but this example illustrates the `reduce` method's mechanisms.

4.4 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 kdt
dir(kdt)
dir(kdt.DiGraph)
dir(kdt.DeVec)
dir(kdt.SpVec)
```

Entity	Name	Elements	Comments
Module	KDT	version, revision, sendFeedback, master	Methods unrelated to any specific data structure
Class	Vec		Distributed parallel vector (dense and sparse)
		Vec, [], =, +, -, += , -=, <>, >, findInds, abs, any, all, nnn, addVFilter, …	Methods
Class	DiGraph		Directed graph
		DiGraph, generateRMAT, 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
		addEFilter, delEFilter	Utility routines
		In, Out	Constants
	Mat		Distributed Sparse Matrix
		Mat, load, ones, spOnes	Constructors
		SpGEMM, SpMV, toVec, eye, transpose, removeMainDiagonal, toBool	Linear algebraic primitives
		nvert, nedge, +, -, *, /, max, min, sum, scale, apply, reduce	General-purpose routines

Table 1. KDT library hierarchy

4.5 Graph Generators

The `DiGraph` class includes the graph generators in this section. Currently, edges may be created as a `Vec` object directly with a subsequent call to the `DiGraph` method, by the `load` or `UFget` method, by a constructor such as `fullyConnected`, by creating edges in a `Mat` class instance, or by using an application-specific method like `generateRMAT`.

4.5.1 generateRMAT

The `generateRMAT` method 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 = kdt.DiGraph.generateRMAT(scale, edgeFactor=16,
initiator=[0.57,0.19,0.19,0.05], delIsolated=True,
retKernel1Time=False, element=True)
```

Description

The `generateRMAT` method creates an input graph as defined by the Graph500 benchmark. The required input parameter `scale` (integer 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. The optional `edgeFactor` argument specifies the average in-/out-degree of the vertices, or half the average total degree, where the number of degrees is before any isolated vertices are deleted (see the `delIsolated` argument below). The optional `initiator` argument is an array of 4 floating-point numbers denoting the probability of a new edge arising in each quadrant of the Kronecker matrix, ordered upper-left, upper-right, lower-left, lower-right. The default value is equal to [0.57, 0.19, 0.19, 0.05], which are the values specified by the Graph500 benchmark. Some vertices may have no edges incident to them; such vertices may deleted by setting the optional `delIsolated` argument to True. The time returned from `generateRMAT` 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 Kernel 1 of 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 = kdt.DiGraph()
time = G.generateRMAT(scale)
```

4.5.2 generate2DTorus

The `generate2DTorus` method creates a `DiGraph` object reflecting the connectivity pattern of a 2D torus.

Syntax

```
G = kdt.generate2DTorus(nnode)
```

Description

The `generate2DTorus` 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 `generate2DTorus(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 = kdt.DiGraph.generate2DTorus(nnodes)
```

4.5.1 generateSelfLoops

The `generateSelfLoops` method creates a `DiGraph` object in which each vertex is directly connected to itself. The edges are inserted into a newly created `DiGraph` object, which is the return value from the method.

Syntax

```
G = kdt.generateSelfLoops(nvert)
```

Description

The `generateSelfLoops` method creates a `DiGraph` with each vertex having a directed edge to 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 itself.

```
G = kdt.DiGraph.generateSelfLoops(nvert)
```

4.5.2 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 = kdt.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 = kdt.DiGraph.fullyConnected(nvert)
```

4.6 I/O

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

4.6.1 `load`: Reading a graph from a file

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

Syntax

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

```
G = kdt.0.2.load(fname)
```

Description

The `load` method loads a file in the Coordinate Format of the Matrix Market Exchange Formats [MatrixMarket] directly into a `DiGraph` or `0.2` object.

Version 0.2

The source-vertex / destination-vertex (for `DiGraph`) or edge-number / vertex-number (for `0.2`) 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 v0.2, the error-checking of a source-vertex or destination-vertex or edge-number or vertex-number 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 = kdt.DiGraph.load('mymatrix.mtx')
```

4.6.2 `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')
```

4.6.3 `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.

Note: See Errata #3 in section 6.5.

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 = kdt.DiGraph.UFget('Andrianov/ex3sta1')
```

4.6.4 `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.

Note: Printing of `DiGraph` or (sparse or dense) `Vec` instances will, in general, require participation of all the active processes, and thus cannot be done solely on the master, and thus should not be done inside a `master` check. Doing so typically results in the program hanging.

Syntax

```
bool = kdt.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` script in KDT, restricts printing output just to the master process.

```
    if kdt.master():
            print 'Graph500 benchmark run for scale = %2i' % scale
            print 'Kernel 1 time = %8.4f seconds' % K1elapsed
            print "\nKernel 2 BFS execution times"
            printstats(K2elapsed, "time", False)
```

4.7 Advanced Usage

KDT v0.2 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, v0.2 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 = kdt.DiGraph()
shew.prot2prot = kdt.DiGraph.load(‘shewanella/protein_protein.mtx’)
shew.DNA2prot = kdt.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.

5 Performance

KDT is intended to enable subject-matter experts who are not graph experts to solve large problems quickly, including both the development and execution phases of a program, and thus performance is a vital aspect of KDT. Its performance has not yet been fully characterized. Performance of specific methods, where available, is provided in their descriptions.

Today we have no means to execute KDT interactively in parallel, where "interactive" means having the full range of IPython's abilities (*e.g.,* breakpoints, stepping, variable inspection) at the user's disposal. Thus we consider interactive and parallel performance separately.

5.1 Interactive

Since interactive execution means serial execution and memory capacities of a single node for v0.2, users will likely limit the problem sizes they develop with interactively to get quick response times.

The Graph500 example in KDT (see `examples/Graph500.py`) is configurable for problem size and number of iterations. For scale=15 (equating to the number of graph vertices being 2**15 or 32K), the Graph500 kernel2 itself runs for a single starting vertex in about 0.3 seconds on a modern x86 core. The entire processing of the graph, including validation of the resulting BFS tree, takes about 3 seconds per iteration (starting vertex).

Betweenness centrality (*e.g.*, the class `CentralityTests` in file `test/TestDiGraph.py`) in KDT is also configurable for problem size and sample rate (for approximate BC). Using the `generate2DTorus` graph generator with the number of torus nodes equal to 32 (equating to the number of graph vertices being 1024), exact BC runs in about 8 seconds on a modern x86 core, and for this scale approximate BC is only slightly faster. Note that a 2D torus is a poor performance case for BC, as one factor in BC's computational complexity is the diameter of the graph, which for a 2D torus is the square root of the number of graph vertices, whereas typical RMAT graphs have a diameter more like the log of the number of graph vertices.

5.2 Parallel

The parallel testing of KDT has occurred on three systems:

- Neumann, a 32-core cluster at UCSB
- Triton, a medium-sized 256-node cluster at the San Diego Supercomputer Center (configuration)
- Hopper II, a large 6392-node, 24 cores-per-node Cray XE6 system at the National Energy Research Supercomputer Center (configuration)

The performance characterization of KDT at large scale is still in process. The following performance has already been demonstrated.

Much of the early performance work with KDT has focused on the Graph500 benchmark. In Table 1, initial Graph500 performance numbers from the Triton cluster at the San Diego Supercomputer Center are provided. The units are millions of traversed-edges per second (TEPS); the first number is for each data element being a 64-bit double-precision value; the second is for each data element being a Boolean value; the performance for Boolean values is often a factor of 5-10 faster than 64-bit double-precision values. On 256 cores, the KDT implementation of Graph500 performs at over 1 GTEPS.

Scale	20	23

Number of cores		
16	- / 200.5	25.9 / 171.4
64	- / 518.5	
256	- / 485.1	- / 1,133.2

Table 1. Graph500 performance

Betweenness centrality is a much more compute-intensive applet than the Graph500, so the problem sizes run so far are much smaller. On Neumann a 2D torus with 128 torus nodes (equating to 16K graph vertices) runs exact BC in 1163 seconds and approximate BC (sample rate of 0.05) in 884 seconds on 16 cores.

6 Practicalities

6.1 Downloading

To get the latest released package, go to `http://kdt.sourceforge.net/download.html` and download it.

6.2 Linux

6.2.1 System Requirements

We recommend Python 2.6 or newer; our recent testing has occurred with Python 2.6.5, though KDT v0.2 has been tested somewhat with Python 2.4. KDT has been tested primarily with OpenMPI 1.4 and IPython 0.8.4, though more recently with IPython 0.10.

6.2.2 Installation and Build

The information in this section is repeated verbatim from the README`.txt` file in the base directory of the KDT package. In case of any discrepancies between this text and README`.txt`, README`.txt` should prevail.

KDT is distributed as a Python `distutils` package. It requires the MPI compiler to be specified in the $CC and $CXX environment variables. For example, in `bash`:

```
export CC=mpicxx
export CXX=mpicxx
```

The build and installation is performed by the standard `distutils` `setup` command:

```
$ python setup.py build
$ sudo python setup.py install
```

If the build step fails fails see the "Choosing a compiler" section below.

KDT makes use of some features from C++ TR1. If your compiler does not support TR1 then the free Boost C++ library (http://www.boost.org/) supplies the required headers as well. Make sure `boost/` is in your include path. If it is not, you can append the include path with the `-I` switch to the `setup.py` script. For example, if you installed Boost in `/home/username/include/boost`:

```
$ python setup.py build -I/home/username/include
```

The MPI library must be compiled with `-fPIC`. Python modules must be compiled with `-fPIC`, and that includes all libraries that get linked into the module, which includes the MPI library in the case of KDT. If your MPI was not compiled with `-fPIC` then the link step will fail. If this happens, contact your system administrator.

6.2.2.1 Choosing a Compiler

KDT consists of pure Python classes and a C++ extension. This C++ extension, called pyCombBLAS, is MPI code and must be compiled as such. However, it is also a Python module, so must be compiled with a compiler that is compatible with your Python installation. By default, `distutils` (i.e. `setup.py`) will use the same compiler with which Python was compiled. That was probably not an MPI compiler, so the proper compiler must be specified to `distutils` via the `CC` and `CXX` environment variables.

Note that your system may give you a choice between GNU and Portland Group (PGI) compilers. These are not fully binary compatible with each other, so you must use the same one that your Python was compiled with. Note that `mpicxx` is often just a wrapper around GNU or PGI compilers.

6.2.2.2 System Libraries

If you chose to use a non-default MPI compiler then be aware that the runtime may link to the default MPI libraries anyway. This is probably not what you want. The result may be something like this:

```
ImportError: ./kdt/_pyCombBLAS.so: undefined symbol: _ZN3MPI3Win4FreeEv
```

The solution is to set your LD_LIBRARY environment variable such that the MPI library you compiled with appears before the incorrect defaults.

6.2.2.3 Building without Distutils

If the `setup.py` script fails for you, you can build the C++ extension manually. The `kdt/pyCombBLAS` directory contains a Makefile (named `Makefile-dist`; rename as necessary) which you can tune for your system. The important things to change are:

- `COMPILER` - compiler to use.
- `INCADD` - include your Python build directory as well as your Boost installation if you need it.
- `OPT` - any flags you wish to change.

Once pyCombBLAS is built, you may manually install the `kdt/` directory in your systemwide Python `site-packages` directory, or simply set your `PYTHONPATH` environment variable to point to the parent directory of `kdt/` (*i.e.*, the base of the distribution).

6.2.3 Tests and Examples

The KDT package includes `test/` and `example/` directories at the base level of the distribution, and in particular the `example/Graph500.py` script runs the Graph500 benchmark, which may be a good way to exercise KDT after installation. Because these files embody the use of KDT functionality, not the core functionality itself, they are not installed in the standard `site-packages` directory with the core functionality. A user who downloads KDT for her own use may just leave them in the directory where the package was unzipped/untarred, or may put them in a system-wide directory for others' use as well. For the rest of this section the `test/` and `example/` directories are assumed to reside in a directory pointed to by the `KDTINSTALL` enviroment variable.

The `test/` directory contains four class-specific files: `TestVec.py`, `TestSpVec.py`, `TestDiGraph.py`, and `TestMat.py` , which test the classes with the corresponding names. The current failures are noted in section 6.5 Errata.

The `examples/` directory contains the `Graph500.py` script, which implements the Graph500 benchmark specification, including all validation steps.

6.2.4 Execution with Python or IPython

This section provides details about how to execute KDT scripts via IPython and Python.

6.2.4.1 IPython

KDT has been used interactively only with IPython, namely version 0.8.4, on a single core. The command-line and IPython commands to invoke it follow (note the same information appears in section 2).

```
[sam@neumann ~]$ PYTHONPATH=”$PYTHONPATH:$KDTINSTALL/examples”
[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 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 1.6
Raw input log  : False       #      “”
Timestamping   : False       #      “”
State          : active      #      “”
```

```
Generating a Graph500 RMAT graph with 2^15 vertices...
Duplicates removed (or summed): 78590 and self-loops removed: 0
Generation took 0.371470s.
iteration 1: start=9171, BFS took 0.030166s, verification took
2.803589s and succeeded, TEPS=29,545,933
iteration 2: start=12434, BFS took 0.030009s, verification took
2.762023s and succeeded, TEPS=29,700,392
iteration 3: start=20292, BFS took 0.029619s, verification took
2.818182s and succeeded, TEPS=30,091,517
iteration 4: start=29823, BFS took 0.029394s, verification took
2.728882s and succeeded, TEPS=30,321,679
     […]
iteration 63: start=14394, BFS took 0.030498s, verification took
2.766191s and succeeded, TEPS=29,224,184
iteration 64: start=7008, BFS took 0.031285s, verification took
2.771903s and succeeded, TEPS=28,489,226
Graph500 benchmark run for scale = 15
Kernel 1 time =   0.3715 seconds

Kernel 2 BFS execution times
            min_time: 2.93109416961669922e-02
  firstquartile_time: 3.01442742347717285e-02
         median_time: 3.08154821395874023e-02
  thirdquartile_time: 3.10997962951660156e-02
            max_time: 3.19378376007080078e-02
           mean_time: 3.06311249732971191e-02
         stddev_time: 6.26722467712359749e-04

Kernel 2 number of edges traversed
            min_nedge: 8.91280000000000000e+05
  firstquartile_nedge: 8.91280000000000000e+05
         median_nedge: 8.91280000000000000e+05
  thirdquartile_nedge: 8.91280000000000000e+05
            max_nedge: 8.91280000000000000e+05
           mean_nedge: 8.91280000000000000e+05
         stddev_nedge: 0.00000000000000000e+00

Kernel 2 TEPS
            min_TEPS: 2.79067108782669082e+07
  firstquartile_TEPS: 2.86587086147099845e+07
         median_TEPS: 2.89231245181400143e+07
  thirdquartile_TEPS: 2.95671458547492772e+07
            max_TEPS: 3.04077572545734048e+07
  harmonic_mean_TEPS: 2.90972009933353513e+07
harmonic_stddev_TEPS: 1.99059058009882137e+04

In [2]:
```

6.2.4.2 Python, not via a job manager

KDT has been used in parallel only with the Python language processor, namely Python 2.4.3. The MPI used was OpenMPI 1.4. The Linux version was 2.6.18.

Note: The current Combinatorial BLAS, on which KDT is based, only supports core counts that are perfect squares, meaning that the argument to the `mpirun –n` command must be a perfect square.

For parallel execution with Python and MPI,you can use the following command-line.

```
 [sam@neumann test]$ mpirun -n 4 python $KDTINSTALL/examples/Graph500.py
Generating a Graph500 RMAT graph with 2^15 vertices...
Duplicates removed (or summed): 78883 and self-loops removed: 0
Generation took 0.080711s.
iteration 1: start=12627, BFS took 0.011984s, verification took
0.619967s and succeeded, TEPS=74,322,165
iteration 2: start=20502, BFS took 0.011391s, verification took
0.601116s and succeeded, TEPS=78,192,512
iteration 3: start=8839, BFS took 0.011701s, verification took
0.616723s and succeeded, TEPS=76,119,720
iteration 4: start=1999, BFS took 0.011484s, verification took
0.612136s and succeeded, TEPS=77,559,400
iteration 5: start=71, BFS took 0.011390s, verification took 0.619576s
and succeeded, TEPS=78,199,059
      […]
iteration 62: start=32066, BFS took 0.011214s, verification took
0.605297s and succeeded, TEPS=79,427,725
iteration 63: start=27903, BFS took 0.011867s, verification took
0.608496s and succeeded, TEPS=75,055,323
iteration 64: start=16201, BFS took 0.011442s, verification took
0.609218s and succeeded, TEPS=77,843,838
Graph500 benchmark run for scale = 15
Kernel 1 time =   0.0807 seconds

Kernel 2 BFS execution times
            min_time: 1.09930038452148438e-02
  firstquartile_time: 1.14405155181884766e-02
         median_time: 1.16879940032958984e-02
  thirdquartile_time: 1.18589401245117188e-02
            max_time: 1.21469497680664062e-02
           mean_time: 1.16281397640705109e-02
         stddev_time: 2.91774617905905386e-04

Kernel 2 number of edges traversed
            min_nedge: 8.90685000000000000e+05
  firstquartile_nedge: 8.90685000000000000e+05
```

```
          median_nedge: 8.90685000000000000e+05
   thirdquartile_nedge: 8.90685000000000000e+05
             max_nedge: 8.90685000000000000e+05
            mean_nedge: 8.90685000000000000e+05
          stddev_nedge: 0.00000000000000000e+00

Kernel 2 TEPS
             min_TEPS: 7.33258156991442293e+07
   firstquartile_TEPS: 7.51066276284680367e+07
          median_TEPS: 7.62052171539593935e+07
   thirdquartile_TEPS: 7.78535725269949883e+07
             max_TEPS: 8.10228953379023224e+07
   harmonic_mean_TEPS: 7.65973765427299738e+07
harmonic_stddev_TEPS: 6.64094336990017182e+04
[sam@neumann test]$
```

6.2.4.3 Python, via a job manager

Many large HPC systems require programs using a large number of cores to be submitted via a job management system. The script below works on SDSC's Triton cluster, using the TORQUE (formerly PBS) job management system.

```
     $ qsub testscriptRun.sh
```

where `testscriptRun.sh` looks like

```
#!/bin/bash
#PBS -q batch
#PBS -N pyCombBLAS
#PBS -l nodes=72:ppn=8
#PBS -l walltime=1:20:00
#PBS -o out.testscript23-576-b
#PBS -e err.testscript23-576-b
#PBS -V
#PBS -M alugowski@gmail.com
#PBS -m abe
#PBS -A alugowski-ucsb

export LD_LIBRARY_PATH=/opt/openmpi/gnu/mx/lib:$LD_LIBRARY_PATH

cd ~/trunk/kdt/pyCombBLAS
/opt/openmpi/gnu/mx/bin/mpirun -v -machinefile $PBS_NODEFILE -np 576
python ./testscript.py -1 /home/alugowski-ucsb/matrices/rmat23.mtx
```

6.3 Windows

6.3.1 System Requirements

KDT has been tested with Windows HPC Server 2008 R2, Python 2.7.1, and MPICH2 1.3.2p1.

6.3.2 Installation and Build

The information in this section is repeated verbatim from the `README_win.txt` file in the base directory of the KDT package. In case of any discrepancies between this text and `README_win.txt`, `README_win.txt` should prevail.

KDT is distributed as a Python Distutils package. The commands to build and install it via the typical "python.exe setup_win.py build" approach are encapsulated in the PowerShell script `install_prereqs_KDT.ps1`.

KDT can be installed with the following steps:

1) The install procedure assumes that Python 2.7 and MPICH2 1.3.2p1 are not already installed. If they are, you may wish to ignore the steps related to them and edit `install_prereqs_KDT.ps1` not to operate on those files.

2) Download the Python 2.7 MSI installer from here and the MPICH2 1.3.2p1 MSI installer from here.

3) Download the `KDT_0.2.zip` file onto the head node of the HPC Server cluster into the same directory as the Python and MPICH2 MSI installers.

4) Unzip the KDT file, either with a utility or with the following PowerShell commands

```
PS > $shell_app=new-object -com shell.application
PS > $zip_file=$shell_app.namespace((Get-Location).Path+"\kdt_0.2.zip")
PS > $zip_dest=$shell_app.namespace((Get-Location).Path+"\kdt_0.2")
PS > $zipdest.Copyhere($zip_file.items())
```

5) Use `clusrun` to install (Python 2.7 and MPICH2 and) KDT on the head node and all the cluster nodes. *I.e.*, when executing on the head node,

```
PS > clusrun PowerShell -ExecutionPolicy bypass
    -file \\<headnode-name>\<path>\kdt_0.2\install_prereqs_KDT.ps1
```

It may be worth executing clusrun on a single node to ensure it's working properly before executing it on all the nodes.

For those administrators wishing more control, the build and installation may be performed (on each node) by the standard Distutils setup commands executed in the correct directory, *i.e.*,

```
$ C:\python27\python.exe setup_win.py build
$ C:\python27\python.exe setup_win.py install
```

The second command above must be performed with Administrator privileges.

KDT must be built with the same compiler used to build the MPICH2 library, which is currently the Visual Studio 2008 64-bit compiler. The default build process handles this implicitly. If the build step fails see the "Choosing a compiler" section below.

6.3.2.1 Choosing a Compiler

KDT consists of pure Python classes and a C++ extension. This C++ extension, called pyCombBLAS, is MPI code and must be compiled as such. However, it is also a Python module, so must be compiled with a compiler that is compatible with your Python installation. By default, `distutils` (i.e. `setup_win.py`) will use the same compiler with which Python and MPICH2 were compiled, but if that doesen't happen properly problems can result.

6.3.3 Tests and Examples

The KDT package includes `test/` and `example/` directories at the base level of the distribution, and in particular the `example/Graph500.py` script runs the Graph500 benchmark, which may be a good way to exercise KDT after installation. Because these files embody the use of KDT functionality, not the core functionality itself, they are not installed in the standard `site-packages` directory with the core functionality. A user who downloads KDT for her own use may just leave them in the directory where the package was unzipped/untarred, or may put them in a system-wide directory for others' use as well. For the rest of this section the `test/` and `example/` directories are assumed to reside in a directory pointed to by the `KDTINSTALL` enviroment variable.

The `test/` directory contains four class-specific files: `TestVec.py`, `TestSpVec.py`, `TestDiGraph.py`, and `TestMat.py`, which test the classes with the corresponding names. The current failures are noted in section 6.5 Errata.

The `examples/` directory contains the `Graph500.py` script, which implements the Graph500 benchmark specification, including all validation steps.

6.3.4 Execution with Python or IPython

This section provides details about how to execute KDT scripts via IPython and Python.

6.3.4.1 IPython

KDT has been used extensively with IPython on Linux, but not on Windows. Much of the following section is probably relevant to the execution of IPython on Windows, but has not been exercised, so the rest of this section is stippled.

KDT has been used interactively only with IPython, namely version 0.8.4, on a single core. The command-line and IPython commands to invoke it follow (note the same information appears in section 2).

```
[sam@neumann ~]$ PYTHONPATH="$PYTHONPATH:$KDTINSTALL/examples"
[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 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 1.6
Raw input log  : False      #     ""
Timestamping   : False      #     ""
State          : active     #     ""

Generating a Graph500 RMAT graph with 2^15 vertices...
Duplicates removed (or summed): 78590 and self-loops removed: 0
Generation took 0.371470s.
iteration 1: start=9171, BFS took 0.030166s, verification took
2.803589s and succeeded, TEPS=29,545,933
iteration 2: start=12434, BFS took 0.030009s, verification took
2.762023s and succeeded, TEPS=29,700,392
iteration 3: start=20292, BFS took 0.029619s, verification took
2.818182s and succeeded, TEPS=30,091,517
iteration 4: start=29823, BFS took 0.029394s, verification took
2.728882s and succeeded, TEPS=30,321,679
     […]
iteration 63: start=14394, BFS took 0.030498s, verification took
2.766191s and succeeded, TEPS=29,224,184
iteration 64: start=7008, BFS took 0.031285s, verification took
2.771903s and succeeded, TEPS=28,489,226
Graph500 benchmark run for scale = 15
Kernel 1 time =   0.3715 seconds

Kernel 2 BFS execution times
            min_time: 2.93109416961669922e-02
  firstquartile_time: 3.01442742347717285e-02
         median_time: 3.08154821395874023e-02
  thirdquartile_time: 3.10997962951660156e-02
            max_time: 3.19378376007080078e-02
           mean_time: 3.06311249732971191e-02
         stddev_time: 6.26722467712359749e-04

Kernel 2 number of edges traversed
            min_nedge: 8.91280000000000000e+05
  firstquartile_nedge: 8.91280000000000000e+05
         median_nedge: 8.91280000000000000e+05
  thirdquartile_nedge: 8.91280000000000000e+05
            max_nedge: 8.91280000000000000e+05
           mean_nedge: 8.91280000000000000e+05
         stddev_nedge: 0.00000000000000000e+00
```

```
Kernel 2 TEPS
            min_TEPS: 2.79067108782669082e+07
  firstquartile_TEPS: 2.86587086147099845e+07
         median_TEPS: 2.89231245181400143e+07
  thirdquartile_TEPS: 2.95671458547492772e+07
            max_TEPS: 3.04077572545734048e+07
  harmonic_mean_TEPS: 2.90972009933353513e+07
harmonic_stddev_TEPS: 1.99059058009882137e+04

In [2]:
```

6.3.4.2 Python, not via a job manager

KDT has been used in parallel only with the Python language processor, namely Python 2.7.1.

Note: The current Combinatorial BLAS, on which KDT is based, only supports core counts that are perfect squares, meaning that the argument to the `mpirun –n` command must be a perfect square.

For parallel execution with Python and MPI, you will typically want to create a machinefile in the format described in the MPICH2 User Guide (link on p. 8), such as in hosts.txt, namely

```
# comment line
hostA:n    # assign n ranks to hostA
hostB:m    # assign m ranks to hostB
```

With such a machinefile in hand, you can use the following command-line:

```
[sam@neumann test]$ "c:\program files\mpich2\bin\mpiexec.exe" -n 4
–machinefile hosts.txt c:\python27\python.exe
$KDTINSTALL\examples\Graph500.py
Generating a Graph500 RMAT graph with 2^15 vertices...
Duplicates removed (or summed): 78883 and self-loops removed: 0
Generation took 0.080711s.
iteration 1: start=12627, BFS took 0.011984s, verification took
0.619967s and succeeded, TEPS=74,322,165
iteration 2: start=20502, BFS took 0.011391s, verification took
0.601116s and succeeded, TEPS=78,192,512
iteration 3: start=8839, BFS took 0.011701s, verification took
0.616723s and succeeded, TEPS=76,119,720
iteration 4: start=1999, BFS took 0.011484s, verification took
0.612136s and succeeded, TEPS=77,559,400
iteration 5: start=71, BFS took 0.011390s, verification took 0.619576s
and succeeded, TEPS=78,199,059
      […]
iteration 62: start=32066, BFS took 0.011214s, verification took
0.605297s and succeeded, TEPS=79,427,725
iteration 63: start=27903, BFS took 0.011867s, verification took
0.608496s and succeeded, TEPS=75,055,323
```

```
iteration 64: start=16201, BFS took 0.011442s, verification took
0.609218s and succeeded, TEPS=77,843,838
Graph500 benchmark run for scale = 15
Kernel 1 time =   0.0807 seconds

Kernel 2 BFS execution times
            min_time: 1.09930038452148438e-02
  firstquartile_time: 1.14405155181884766e-02
         median_time: 1.16879940032958984e-02
  thirdquartile_time: 1.18589401245117188e-02
            max_time: 1.21469497680664062e-02
           mean_time: 1.16281397640705109e-02
         stddev_time: 2.91774617905905386e-04

Kernel 2 number of edges traversed
            min_nedge: 8.90685000000000000e+05
  firstquartile_nedge: 8.90685000000000000e+05
         median_nedge: 8.90685000000000000e+05
  thirdquartile_nedge: 8.90685000000000000e+05
            max_nedge: 8.90685000000000000e+05
           mean_nedge: 8.90685000000000000e+05
         stddev_nedge: 0.00000000000000000e+00

Kernel 2 TEPS
            min_TEPS: 7.33258156991442293e+07
  firstquartile_TEPS: 7.51066276284680367e+07
         median_TEPS: 7.62052171539593935e+07
  thirdquartile_TEPS: 7.78535725269949883e+07
            max_TEPS: 8.10228953379023224e+07
  harmonic_mean_TEPS: 7.65973765427299738e+07
harmonic_stddev_TEPS: 6.64094336990017182e+04
[sam@neumann test]$
```

6.4 Debugging

The KDT developers have developed all their Python code interactively on a single core with IPython and then executed it in parallel with Python. When parallel execution encounters bugs that don't appear serially, the primary debugging tool has been printing, either via Python's `print` statement or via the `Vec` `printAll` methods.

6.5 Errata

The following bugs or anomalies are known to exist in v0.2:

1. The definition of the distinction between nonnull and nonzero elements in `DiGraph` and sparse `Vec` instances is not completely coherent. The most glaring instance of this is `Vec` instances that represent indices, in which zero is a valid nonnull element. Because there is not a

sparse `Find` method exposed from the Combinatorial BLAS, KDT converts a sparse `Vec` instance to a dense `Vec` instance before calling the dense `Find` from the Combinatorial BLAS. This conversion obliterates the distinction between a nonnull element whose value is zero and a null element.

2. The `Vec` `print` method will sometimes cause the message
   ```
   ADIOI_UFS_OPEN (line 83): **io Too many open filesMPI_FILE_CLOSE
   (line 51): **iobadfh
   Problem reading binary input file
   ```
   to be printed on stderr and stdout. This appears to effect only the output itself and not the correct functioning of the rest of the program.

3. The `DiGraph` `UFget` method does not act properly on files that have the Matrix Market keyword `symmetric` in their first (comment) line, so the resulting graphs only have half as many edges as expected. The `Pajek/dictionary28` file illustrates this problem. Such a file can still be loaded properly by the following code:
   ```
   G = kdt.UFget('Pajek/dictionary28')
   G2 = G.copy()
   G2.reverseEdges()
   G = G+G2
   ```

4. In addition, a handful of unit tests (see files `TestVec.py`, `TestSpVec.py`, and `TestDiGraph.py` in directory `test`) fail:

   a. Two tests expose that KDT does not always collapse multiple edges between the same source and destination to a single edge; `TestDiGraph:test_load` and `TestDiGraph:test_DiGraph_duplicates`.

   b. `test_UFget_simple` exposes the issue noted in item #3 above

5. On Windows, KDT programs that complete successfully will typically appear to fail with messages like the following:

   ```
   job aborted:
   rank: node: exit code[: error message]
   0: CN01.clus.private.local: 0: process 0 exited without calling finalize
   1: CN01.clus.private.local: 0: process 1 exited without calling finalize
   2: CN01.clus.private.local: 0: process 2 exited without calling finalize
   3: CN01.clus.private.local: 0: process 3 exited without calling finalize
   4: CN02.clus.private.local: 0: process 4 exited without calling finalize
   5: CN02.clus.private.local: 0: process 5 exited without calling finalize
   6: CN02.clus.private.local: 0: process 6 exited without calling finalize
   7: CN02.clus.private.local: 0: process 7 exited without calling finalize
   8: CN03.clus.private.local: 0: process 8 exited without calling finalize
   9: CN03.clus.private.local: 0: process 9 exited without calling finalize
   10: CN03.clus.private.local: 0: process 10 exited without calling finalize
   11: CN03.clus.private.local: 0: process 11 exited without calling finalize
   12: CN04.clus.private.local: 0: process 12 exited without calling finalize
   ```

```
13: CN04.clus.private.local: 0: process 13 exited without calling finalize
14: CN04.clus.private.local: 0: process 14 exited without calling finalize
15: CN04.clus.private.local: 0: process 15 exited without calling finalize
```

6. On Windows, killing the execution of mpiexec via <ctl>C sometimes does not cleanly terminate the execution of all ranks on all nodes, and sometimes leaves Python-language files (*e.g.*, "`xyz.py`") locked.

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 columns and To vertices as rows. The Combinatorial BLAS' C++ interfaces are wrapped into Python as the `pyCombBLAS` module, with classes `pyDenseParVec` and `pySpParVec` (that correspond directly with KDT's `Vec` class), and `pySpParMat` (corresponding to both the `DiGraph` and `Mat` 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 v0.2 is creating a breadth-first search tree from a graph and a root vertex. The code for this is found in `kdt/Algorithms.py`.

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, with a graph and its dual adjacency matrix, which is transposed to work properly with matrix-vector multiplication.

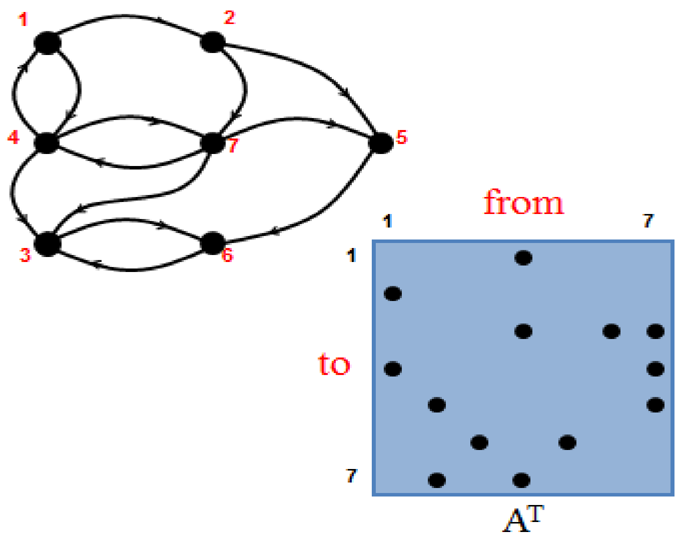

Figure 2. Illustration of bfsTree algorithm (initial state)

Let's make vertex 1 the root of the BFS tree and take the first step of the algorithm in Figure 3. 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.)

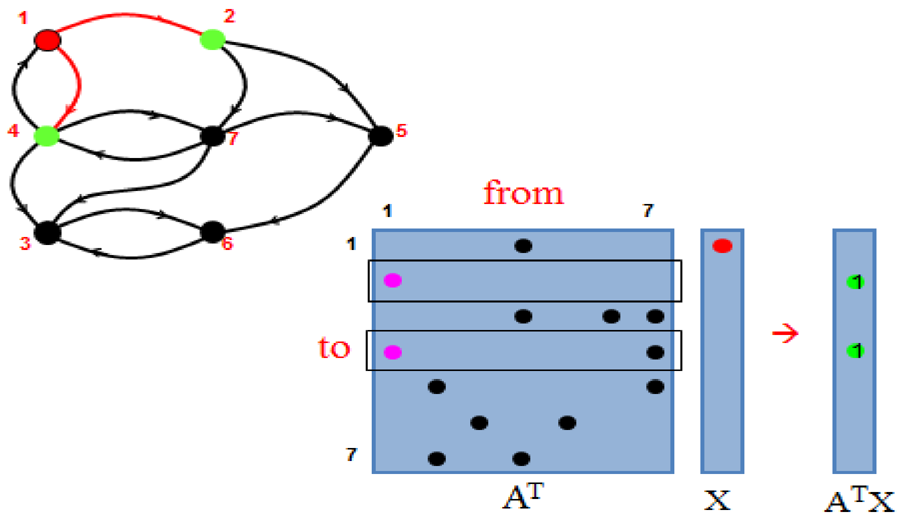

Figure 3. Illustration of bfsTree algorithm (after first iteration)

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 4, 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.

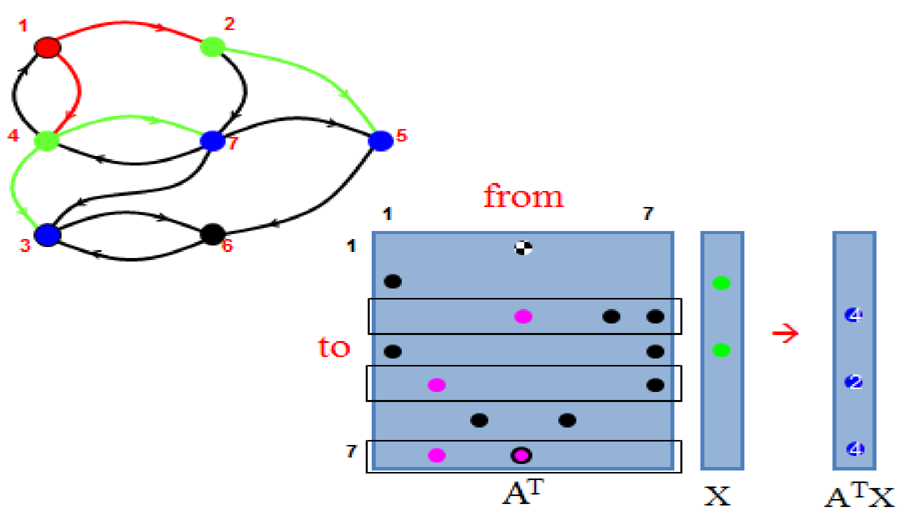

Figure 4. Illustration of bfsTree algorithm (after second iteration)

The next step of the algorithm for this graph is illustrated in Figure 5. 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.

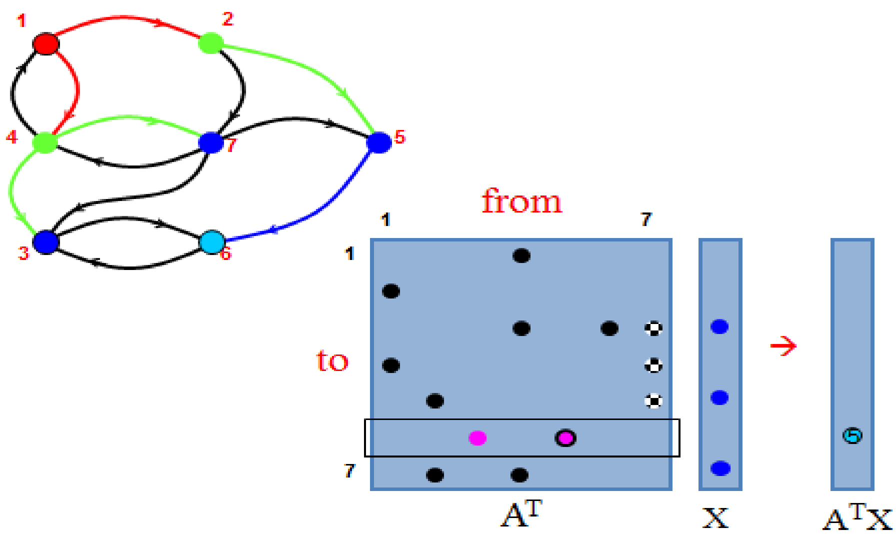

Figure 5. Illustration of bfsTree algorithm (after third iteration)

The last 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 [0, 1, 1, 4, 1, 2, 3, 4]; note that the root vertex is its own parent and that in KDT vertices are numbered from 0, even though vertex 0 has no edges in this example.

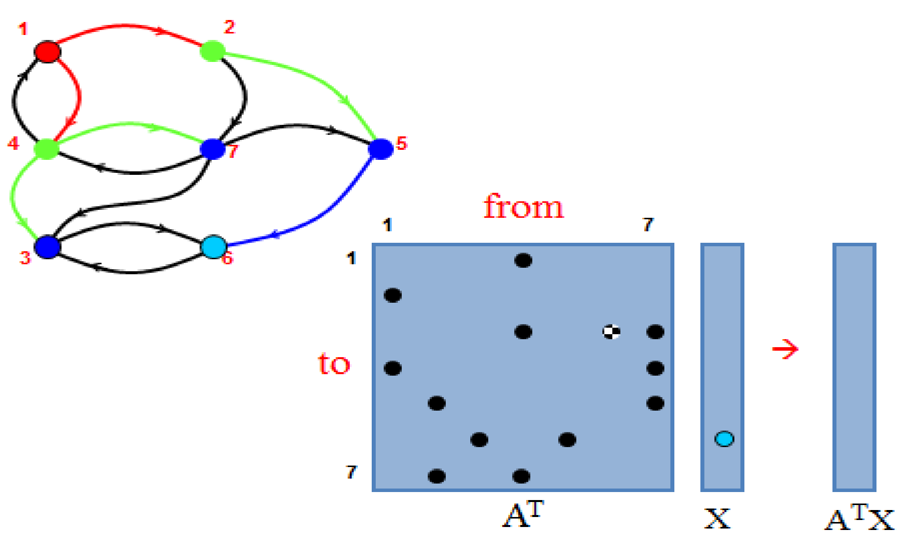

Figure 6. Illustration of bfsTree algorithm (after fourth iteration)

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`. Eventually this operation will be generalized in the style of the `Reduce` 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-/in-edge of a DiGraph by the corresponding element of a sparse `Vec`, is implemented for the in-edge case as

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

A.2.4 Reducing the out-edges (rows) 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 row-wise sum of the absolute values of the nonnull elements could be implemented by

```
ret = self.spm.Reduce(pcb.pySpParMat.Row(),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 simple edges connecting pairs of vertices.

0.2: A collection of vertices and undirected edges, where any edge may connect any subset of 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

[Page] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, "The PageRank Citation Ranking: Bringing Order to the Web", 1998, http://www-db.stanford.edu/~backrub/pageranksub.ps.

[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
R0.07	February 21, 2011	Add pageRank, toBool. Remove cluster. Fix several minor additions/deletions/errors.
R0.08	February 21, 2011	Add normalizeEdgeWeights, change names to spOnes/sprang, note that only serial work can be done inside a master() if-check.
R0.09	February 28, 2011	Clean up out-/in-edge directions and ParVec/SpParVec indexing modes.
R0.10	March 16, 2011	Add high-level note about SPMD execution; add note about UFget oddities; change SpParVec:__getitem__ to reflect CombBLAS changes; add sort/sorted/topK for ParVec/SpParVec; add mean/std for SpParVec. Include last directory clean-up before v0.1 release and errata.
R0.11	April 13, 2011	Several typos

R0.12	August 2, 2011	Update to v0.1W content, namely the 0.2 class and Windows installation instructions.
R0.13	October 11, 2011	Update with semantic graph content, new Vec and Mat classes, delete references to HyGraph class (which has not been tested post-rework).