The Knowledge Discovery Toolbox (Version 0.1) User Guide

Last change date : January 22, 2011 **Revision : r0.03**

Open issues:

- Add start-up instructions (from shell)
- Examples for centrality() and cluster()
- no FP:
- Centrality() needs scale factor because of no FP
- Describe find/sparse (G names, that is)
- Define SpParVec

Contents

1 Overview

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

Version 0.1

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

1.1 Context

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

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

Comment [DW1]: Appeared red when printed

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

KDT's complex algorithms are difficult for even graph experts to verify, and so whenever possible KDT supports internal or user-driven verification of results. That variously consists of internal checks in KDT routines, companion routines that can validate a data structure (*i.e.*, `isBfsTree()` can validate the results of `bfsTree()`), and synthetic inputs whose metrics can be analytically derived (*i.e.*, each vertex of the graph created by `gen2DTorusEdges(scale)` has an identical betweenness centrality value of 0.5 * 2^(3*`scale`*0.5) – 2^`scale` + 1 for an even `scale`).

1.2 Intended use cases

Version 0.1

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

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

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

1.2.2 Calculating the centrality of vertices of a graph

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

1.2.3 Clustering the vertices of a graph

Clustering provides insight into which vertices are most associated by some criterion. As with centrality, KDT provides a few algorithms, believed to work best on very large data, that were selected from many that have been proposed or implemented.

1.3 Deferred use cases

Version 0.1

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

1.4 Legend and Naming Convention

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

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

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

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

1.5 Giving feedback to the KDT developers

KDT includes `sendFeedback` , a built-in feedback method that enables users to type in code that they wish KDT could execute and then send it to the developers. It uses IPython's `%logstart` facility to capture the code snippet.

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

The feedback mechanism can be used as follows:

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

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

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

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

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

2 Graph500 example

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

[ToDo: fill in details of Python startup and running Graph500 script] Starting at a Linux shell prompt on the cluster head-node, with KDT installed in its default location (`/opt/kdt/`[ToDo: confirm]), you can type the following to start a parallel IPython session to run KDT scripts. […]

```
import time
import numpy as np
import scipy as sc
import kdtdm.digraph as kdtdg         # “dm” == distributed memory
#
#          […]
#
#     previous kernels have created the following live variables
#     scale:  the log base 2 of the number of edges in the graph
#     edges:  the edges originally used to build the graph
#     G:       the directed graph
```

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

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

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

```
for i in starts:
      start = deg3verts[i];
      before = time.clock();
      parents = kdtdg.bfsTree(G,start);
      k2Elapsed += time.clock() – before;
      if not k2Validate(G, start, parents):
          print “Invalid BFS tree generated by kdt.bfsTree”;
     k2Edges += len(parents[edges.source()] <> -1).nonzero());
[FIX:’source’]
```

The `k2Validate` method further illustrates the use of KDT with edge and vertex vectors. For instance, one section of code validates that each edge’s endpoints are one level apart in the BFS tree.

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

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, the graph objects are accessible via standard Python methods such as subscripting, comparisons, and `nonzero`.

3 Algorithms, Methods and Classes

Version 0.1

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

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

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

```
     import kdt.DiGraph as kdtdg      [FIX:  ensure path is right]
```

3.1 Algorithms

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

3.1.1 Centrality

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

Syntax

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

Description

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

3.1.1.1 Exact betweenness centrality algorithm

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

Syntax

```
cl = kdtdg.centrality('exactBC', G[, normalize=False]);
```

Description

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

Performance

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

3.1.1.2 Approximate betweenness centrality algorithm

This algorithm 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 = kdtdg.centrality('approxBC', G[, normalize=False, sample=]);
```

Description

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

Performance

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

3.1.2 Single-membership Clustering

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

Syntax

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

Description

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

3.1.2.1 Markov clustering algorithm

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

Syntax

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

Description

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

3.1.2.2 Modularity clustering algorithm

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

Syntax

```
cl = kdtdg.cluster('modularity', G);
```

Description

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

3.1.3 Search

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

3.1.3.1 Breadth-first Search Tree

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

Syntax

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

Description

The BFS function takes as input a `DiGraph` instance and the index of the vertex from which to start the search. It returns a `ParVec` (of length equal to the number of vertices in the graph) that denotes, for each respective vertex of the graph, the vertex's parent in the BFS tree. The `start` vertex's parent is itself. A vertex that is unreachable from the `start` vertex has a parent of -1.

3.2 General-purpose methods

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

3.2.1 Built-in methods for ParVec vectors

Version 0.1

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

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

Syntax

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

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

v[k] = m
m = v[k]
v[vec] = m            # vec an integer vector, m an integer scalar

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

Description

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

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

3.2.2 Non-built-in methods for ParVec vectors

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

Syntax

```
# v and v2 are ParVec instances, m a scalar
v2 = v.copy()         # I.e., simple assignment does not copy

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

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

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

v = kdtdg.ParVec.ones(size);
v = kdtdg.ParVec.zeros(size);
v = kdtdg.ParVec.range([start,] stop)
v2 = v.findInds()

m = v.sum();

v.randPerm();
```

Description

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

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

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

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

3.2.3 Non-built-in methods for DiGraph objects

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

Version 0.1

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

Syntax

```
# G and G2 are DiGraphs
G2 = G.copy()         # I.e., simple assignment (“=”) does not copy
k = G.nvert()         # number of vertices
k = G.nedge()         # number of edges
```

Description

These methods have their obvious definitions.

3.2.3.1 degree

The degree method of the DiGraph class calculates the degree of each vertex.

Syntax

```
degrees = G.degree([dir=InOut])
```

Description

The degree method calculates the degree (number of incident edges) of each vertex, returning a ParVec object. The optional dir argument specifies whether it is InOut (default), In, or Out degree.

Example

The code below calculates the in-degree of the vertices of a DiGraph.

```
degrees = G.degree(kdtdg.In());
```

3.2.4 Advanced methods for DiGraph objects

The DiGraph class implements several advanced methods.

3.2.4.1 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

[**FIX**: worth having In/Out/InOut flavors?]

The neighbor method of the DiGraph class calculates, for a ParVec of input starting vertices, which vertices are neighbors; *i.e.*, connected by edges. The dir argument determines whether it calculates the neighborhood based on both InOut edges (default), In edges, or Out edges. The nhop argument determines how many hops from the starting vertices are used to calculate the neighbors. The return value is a ParVec with the indices of neighboring vertices.

Example

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

```
neighbors = G.neighbor(start, dir=kdtdg.Out());
```

3.2.4.2 pathHop

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

Syntax

```
[fromV, toV] = G.pathHop(start)
```

Description

[**FIX**: worth having In/Out/(InOut??) flavors?]

For a `ParVec` of input starting vertices, the `pathHop` method calculates which vertices can be reached across a single edge in the `DiGraph` instance and, for each reachable vertex, which starting vertex has an edge to it. The source vertices and the new vertices are returned in `ParVec` and Boolean `ParVec` instances, respectively. The set of reachable vertices may include starting vertices. In the case of more than one start vertex having an edge to a reachable vertex, the highest-numbered vertex is selected. `pathHop` can be used repeatedly to implement a breadth-first search.

Example

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

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

3.2.5 DiGraph

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

Syntax

```
G = kdtdg.DiGraph(edgev)
```

Description

The `DiGraph` method creates a `DiGraph` instance. The required input parameter `edges` is a nested tuple of `ParVec` objects created by the program or by generators such as `genGraph500Edges`. The returned `DiGraph` object is a directed graph. The values of any duplicate edges (same source and destination) are summed in the creation of the `DiGraph` object.

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

3.3 Package and class structure

KDT is structured as shown in Table 1, with some methods omitted for brevity.

Entity	Name	Elements	Comments
Module	kdt.Graph		
Class	ParVec		Distributed parallel vector
		ParVec, [], =, +, -, += , -=, <>, >, findInds, abs, any, all, nnz, …	Methods
Module	kdt.DiGraph		
Class	DiGraph		Directed graph
		DiGraph, genGraph500Graph, load	Constructors
		centrality, cluster	Algorithms
		[], nvert, nedge, InOut	General-purpose routines
		load	I/O
		bfsTree, toEdges	Primitives
Class	ParVec		Inherited from Graph

Table 1. KDT library hierarchy

3.4 Graph Generators

The `kdt.digraph` class includes the graph generators in this section. With v0.1 edges must be created as an `DiEdgeV` object directly, either by the `load` method or by using an application-specific method like `genGraph500Graph`.

3.4.1 `genGraph500Graph`

The `genGraph500Graph` function creates a graph following the specifications for the V1.1 Graph500 benchmark's input graph [Graph500]. The edge vector represents an RMAT graph with specific values provided by the benchmark.

Syntax

```
time = kdtdg.genGraph500Graph(scale[, edgeFactor=None])
```

Description

The `genGraph500Graph` function creates an input graph as defined by the Graph500 benchmark. The required input parameter `scale` (logarithm base 2 of the number of desired vertices) defines the number of vertices. The edges are directed, though each edge has a twin going in the other direction because the specification requires the graph to be symmetric. The optional expert keyword argument `edgeFactor` defines how many edges exist on average for each vertex (and thus is half the average degree of a vertex); the default value is 16, as defined by the Graph500 benchmark. Some vertices may have no edges incident to them. The time returned from `genGraph500Graph` includes the execution time of converting the edge vector to a graph but does not include the time to create the edge vector, as defined by the Graph500 benchmark.

3.5 I/O

The `kdt.digraph` class includes the I/O method in this section.

3.5.1 `load`: Reading a graph from a file

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

Syntax

```
G = kdtdg.DiGraph.load(fname);
```

Description

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

Version 0.1

Since the `DiGraph` class supports only 64-bit integer values for v0.1, any floating-point data in the file will be cast to an integer during the `load`.

Example

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

```
        G = kdtdg.DiGraph.load('mymatrix.mtx');
```

3.6 Toolbox Structure

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

	Module	Class	Methods	Comments
kdtsm	<not expanded here, contents superset of kdtdm>			Serial, in-memory
kdtdm				Distributed, in-memory
	kdt.graph			
		Graph		General Graph

		ParVec		Vertex vector
			len, degree, …	Methods
	kdt.digraph			
		DiGraph		Directed graph
			central, cluster, indegree, …	Methods
	kdt.multigraph			Future Multigraph
	kdt.hypergraph			Future Hypergraph
kdtsd	<not expanded here, contents roughly same as kdtdm>			Future Serial, on disk
kdtdd	<not expanded here, contents roughly same as kdtdm>			Future Distributed, on disk

Table 2 Diverse KDT implementations and graph types

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

```
    import kdtdm.digraph as kdtdg
```

Anticipated future versions could be imported as

```
    import kdtsm.digraph as kdtdg
```

for the single-node in-memory version and

```
    import kdtdd.digraph as kdtdg
```

for a distributed disk-based implementation.

4 Practicalities

4.1 Downloading

[[FIX: include details.]]

4.2 Installation

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

5 Glossary

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

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

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

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

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

6 References

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

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

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

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

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

7 Revision History

R0.01	December 16, 2010	Steve Reinhardt	Initial draft
R0.02	December 20, 2010	Steve Reinhardt	Revisions including comments from Aydin Buluc and Drew Waranis
R0.03	January 22, 2011	Steve Reinhardt	Bring current with code base (almost), documenting neighbors() and pathHop() as well as several other methods.