A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis*

Adam Lugowski† David Alber‡ Aydın Buluç§ John R. Gilbert¶ Steve Reinhardt‖
Yun Teng** Andrew Waranis††

Abstract

The Knowledge Discovery Toolbox (KDT) enables domain experts to perform complex analyses of huge datasets on supercomputers using a high-level language without grappling with the difficulties of writing parallel code, calling parallel libraries, or becoming a graph expert. KDT provides a flexible Python interface to a small set of high-level graph operations; composing a few of these operations is often sufficient for a specific analysis. Scalability and performance are delivered by linking to a state-of-the-art back-end compute engine that scales from laptops to large HPC clusters. KDT delivers very competitive performance from a general-purpose, reusable library for graphs on the order of 10 billion edges and greater. We demonstrate speedup of 1 and 2 orders of magnitude over PBGL and Pegasus, respectively, on some tasks. Examples from simple use cases and key graph-analytic benchmarks illustrate the productivity and performance realized by KDT users. Semantic graph abstractions provide both flexibility and high performance for real-world use cases. Graph-algorithm researchers benefit from the ability to develop algorithms quickly using KDT's graph and underlying matrix abstractions for distributed memory. KDT is available as open-source code to foster experimentation.

Keywords: massive graph analysis, scalability, sparse matrices, open-source software, domain experts

1 Introduction

Analysis of very large graphs has become indispensable in fields ranging from genomics and biomedicine to financial services, marketing, and national security, among others. In many applications, the requirements are moving beyond relatively simple filtering and aggregation queries to complex graph algorithms involving clustering (which may depend on machine learning methods), shortest-path computations, and so on. These complex graph algorithms typically require high-performance computing resources to be feasible on large graphs. However, users and developers of complex graph algorithms are hampered by the lack of a flexible, scalable, reusable infrastructure for high-performance computational graph analytics.

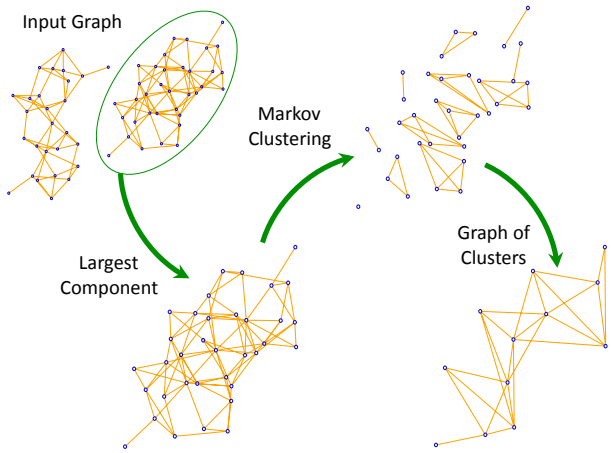

Figure 1: An example graph analysis mini-workflow in KDT.

Our Knowledge Discovery Toolbox (KDT) is the first package that combines ease of use for domain (or subject-matter) experts, scalability on large HPC clusters where many domain scientists run their large scale experiments, and extensibility for graph algorithm developers. KDT addresses the needs both of graph analytics users (who are not expert in algorithms or high-performance computing) and of graph analytics researchers (who are developing algorithms and/or tools for graph analysis). KDT is an open-source, flexible, reusable infrastructure that implements a set of key graph operations with excellent performance on standard computing hardware.

The principal contribution of this paper is the introduction of a graph analysis package which is useful to domain experts and algorithm designers alike. Graph analysis packages that are entirely written in very-high level languages such as Python perform poorly. On the other hand, simply wrapping an existing high-performance package into a higher level language impedes user productivity because it exposes the underly-

*This work was partially supported by NSF grant CNS-0709385, by DOE contract DE-AC02-05CH11231, by a contract from Intel Corporation, by a gift from Microsoft Corporation and by the Center for Scientific Computing at UCSB under NSF Grant CNS-0960316.

†UC Santa Barbara. Email: alugowski@cs.ucsb.edu
‡Microsoft Corp. Email: david.alber@microsoft.com
§Lawrence Berkeley Nat. Lab. Email: abuluc@lbl.gov
¶UC Santa Barbara. Email: gilbert@cs.ucsb.edu
‖Cray, Inc. Email: spr@cray.com
**UC Santa Barbara. Email: yunteng@umail.ucsb.edu
††UC Santa Barbara. Email: andrewwaranis@umail.ucsb.edu

```
# the variable bigG contains the input graph
# find and select the giant component
comp = bigG.connComp()
giantComp = comp.hist().argmax()
G = bigG.subgraph(mask=(comp==giantComp))

# cluster the graph
clus = G.cluster('Markov')

# get per-cluster stats, if desired
clusNvert = G.nvert(clus)
clusNedge = G.nedge(clus)

# contract the clusters
smallG = G.contract(clusterParents=clus)
```

Figure 2: KDT code implementing the mini-workflow illustrated in Figure 1.

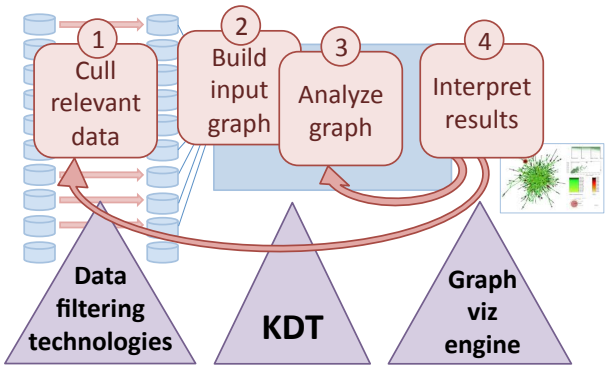

Figure 3: A notional iterative analytic workflow, in which KDT is used to build the graph and perform the complex analysis at steps 2 and 3.

ing package's lower-level abstractions that were intentionally optimized for speed.

KDT uses high-performance kernels from the Combinatorial BLAS [8]; but KDT is a great deal more than just a Python wrapper for a high-performance back-end library. Instead it is a higher-level library with real graph primitives that does not require knowledge of how to map graph operations to a low-level high performance language (linear algebra in our case). It uses a distributed memory framework to scale from a laptop to a supercomputer consisting of hundreds of nodes. It is highly customizable to fit users' problems.

Our design activates a virtuous cycle between algorithm developers and domain experts. High-level domain experts create demand for algorithm implementations while lower-level algorithm designers are provided with a user base for their code. Domain experts use graph abstractions and existing routines to develop new applications quickly. Algorithm researchers build new algorithm implementations based on a robust set of primitives and abstractions, including graphs, dense and sparse vectors, and sparse matrices, all of which may be distributed across the memory of multiple nodes of an HPC cluster.

Figure 1 is a snapshot of a sample KDT workflow (described in more detail in Section 4.6). First we locate the largest connected component of the graph; then we divide this "giant" component of the graph into clusters of closely-related vertices; we contract the clusters into supervertices; and finally we perform a detailed structural analysis on the graph of supervertices. Figure 2 shows the actual KDT Python code that implements this workflow.

The remainder of this paper is organized as follows. Section 2 highlights KDT's goals and how it fits into a graph analysis workflow. Section 3 covers projects related to our work. We provide examples and performance comparisons in Section 4. The high-level language interface is described in Section 5 followed by an overview of our back-end in Section 6. Finally we summarize our contribution in Section 7.

2 Architecture and Context

A repeated theme in discussions with likely user communities for complex graph analysis is that the domain expert analyzing a graph often does not know in advance exactly what questions he or she wants to ask of the data. Therefore, support for interactive trial-and-error use is essential.

Figure 3 sketches a high-level analytical workflow that consists of (1) culling possibly relevant data from a data store (possibly disk files, a distributed database, or streaming data) and cleansing it; (2) constructing the graph; (3) performing complex analysis of the graph; and (4) interpreting key portions or subgraphs of the result graph. Based on the results of step 4, the user may finish, loop back to step 3 to analyze the same data differently, or loop back to step 1 to select other data to analyze.

KDT introduces only a few core concepts to ease adoption by domain experts. The top layer in Figure 4 shows these; a central graph abstraction and high-level graph methods such as `cluster` and `centrality`. Domain experts compose these to construct compact, expressive workflows via KDT's Python API. Exploratory analyses are supported by a menu of different algorithms for each of these core methods (*e.g.*, Markov and even-

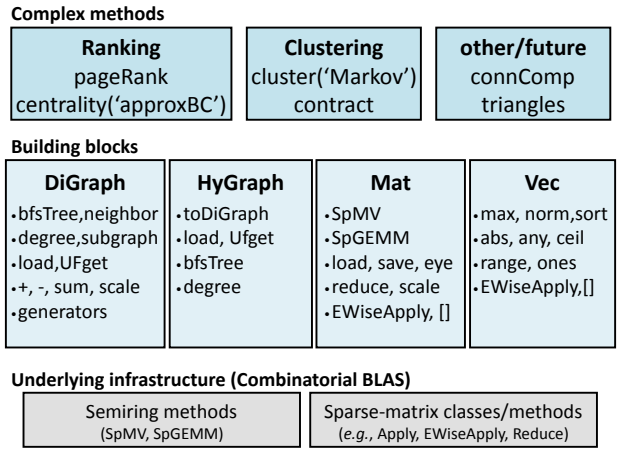

Figure 4: The architecture of Knowledge Discovery Toolbox. The top-layer methods are primarily used by domain experts, and include `centrality` and `cluster` for semantic graphs. The middle-layer methods are primarily used by graph-algorithm developers to implement the top-layer methods. KDT is layered on top of Combinatorial BLAS.

tually spectral and k-means algorithms for clustering). Good characterizations of each algorithm's fitness for various types of very large data are rare and so most target users will not know in advance which algorithms will work well for their data. We expect the set of high-level methods to evolve over time.

The high-level methods are supported by a small number of carefully chosen building blocks. KDT is targeted to analyze large graphs for which parallel execution in distributed memory is vital, and so its primitives are tailored to work on entire collections of vertices and edges. As the middle layer in Figure 4 illustrates, these include directed graphs (`DiGraph`), hypergraphs (`HyGraph`), and matrices and vectors (`Mat`, `Vec`). The building blocks support lower-level graph and sparse matrix methods (for example, `degree`, `bfsTree`, and `SpGEMM`). This is the level at which the graph algorithm developer or researcher programs KDT.

Our current computational engine is Combinatorial BLAS [8] (shortened to CombBLAS), which gives excellent and highly scalable performance on distributed-memory HPC clusters. It forms the bottom layer of our software stack.

Knowledge discovery is a new and rapidly changing field, and so KDT's architecture fosters extensibility. For example, a new clustering algorithm can easily be added to the `cluster` routine, reusing most of the existing interface. This makes it easy for the user to adopt a new algorithm merely by changing the `algorithm` argument. Since KDT is open-source (available at `http://kdt.sourceforge.net`), algorithm researchers can look at existing methods to understand implementation details, to tweak algorithms for their specific needs, or to guide the development of new methods.

3 Related Work

KDT combines a high-level language environment, to make both domain users and algorithm developers more productive, with a high-performance computational engine to allow scaling to massive graphs. Several other research systems provide some of these features, though we believe that KDT is the first to integrate them all.

Titan [35] is a component-based pipeline architecture for ingestion, processing, and visualization of informatics data that can be coupled to various high-performance computing platforms. Pegasus [19] is a graph-analysis package that uses MapReduce [11] in a distributed-computing setting. Pegasus uses a generalized sparse matrix-vector multiplication primitive called `GIM-V`, much like KDT's `SpMV`, to express vertex-centered computations that combine data from neighboring edges and vertices. This style of programming is called "think like a vertex" in Pregel [27], a distributed-computing graph API. In traditional scientific computing terminology, these are all BLAS-2 level operations; neither Pegasus nor Pregel currently includes KDT's BLAS-3 level `SpGEMM` "friends of friends" primitive. BLAS-3 operations are higher level primitives that enable more optimizations and generally deliver superior performance. Pregel's C++ API targets efficiency-layer programmers, a different audience than the non-parallel-computing-expert domain experts (scientists and analysts) targeted by KDT.

Libraries for high-performance computation on large-scale graphs include the Parallel Boost Graph Library [17], the Combinatorial BLAS [8], and the Multithreaded Graph Library [4]. All of these libraries target efficiency-layer programmers, with lower-level language bindings and more explicit control over primitives.

GraphLab [26] is an example of an application-specific system for parallel graph computing, in the domain of machine learning algorithms. Unlike KDT, GraphLab runs only on shared-memory architectures.

4 Examples of use

In this section, we describe experiences using the KDT abstractions as graph-analytic researchers, implementing complex algorithms intended as part of KDT itself (breadth-first search, betweenness centrality, PageRank, Gaussian belief propagation, and Markov clustering), and as graph-analytic users, implementing

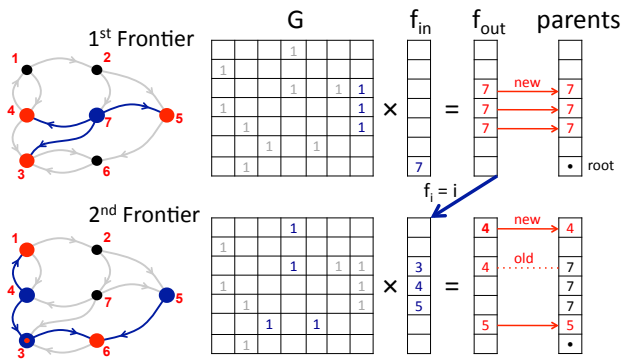

Figure 5: Two steps of breadth-first search, starting from vertex 7, using sparse matrix-sparse vector multiplication with "max" in place of "+".

a mini-workflow.

4.1 Breadth-First Search

4.1.1 An algebraic implementation of BFS Breadth-first search (BFS) is a building block of many graph computations, from connected components to maximum flows, route planning, and web crawling and analysis [31, 15]. BFS explores a graph starting from a specific vertex, identifying the "frontiers" consisting of vertices that can be reached by paths of 1, 2, 3, ... edges. BFS also computes a spanning tree, in which each vertex in one frontier has a parent vertex from the previous frontier.

In computing the next frontier from the current one, BFS explores all the edges out of the current frontier vertices. For a directed simple graph this is the same computational pattern as multiplying a sparse matrix (the transpose of the graph's adjacency matrix) by a sparse vector (whose nonzeros mark the current frontier vertices). The example in Figure 5 discovers the first two frontiers `f` from vertex 7 via matrix-vector multiplication with the transposed adjacency matrix G, and computes the parent of each vertex reached. `SpMV` is KDT's matrix-vector multiplication primitive.

Notice that while the structure of the computation is that of matrix-vector multiplication, the actual "scalar" operations are selection operations not addition and multiplication of real numbers. Formally speaking, the computation is done in a semiring different from $(+, \times)$. The `SpMV` user specifies the operations used to combine edge and vertex data; the computational engine then organizes the operations efficiently according to the primitive's well-defined memory access pattern.

It is often useful to perform BFS from multiple vertices at the same time. This can be accomplished in KDT by "batching" the sparse vectors for the searches into a single sparse matrix and using the sparse matrix-matrix multiplication primitive `SpGEMM` to advance all searches together. Batching exposes three levels of potential parallelism: across multiple searches (columns of the batched matrix); across multiple frontier vertices in each search (rows of the batched matrix or columns of the transposed adjacency matrix); and across multiple edges out of a single high-degree frontier vertex (rows of the transposed adjacency matrix). The Combinatorial BLAS `SpGEMM` implementation exploits all three levels of parallelism when appropriate.

4.1.2 The Graph500 Benchmark The intent of the Graph500 benchmark [16] is to rank computer systems by their capability for basic graph analysis just as the Top500 list [30] ranks systems by capability for floating-point numerical computation. The benchmark measures the speed of a computer performing a BFS on a specified input graph in *traversed edges per second* (TEPS). The benchmark graph is a synthetic undirected graph with vertex degrees approximating a power law, generated by the RMAT [24] algorithm. The size of the benchmark graph is measured by its *scale*, the base-2 logarithm of the number of vertices; the number of edges is about 16 times the number of vertices. The RMAT generation parameters are $a = 0.59, b = c = 0.19, d = 0.05$, resulting in graphs with highly skewed degree distributions and a low diameter. We symmetrize the input to model undirected graphs, but we only count the edges traversed in the original graph for TEPS calculation, despite visiting the symmetric edges as well.

We have implemented the Graph500 code in KDT, including the parallel graph generator, the BFS itself, and the validation required by the benchmark specification. Per the spec, the validation consists of a set of consistency checks of the BFS spanning tree. The checks verify that the tree spans an entire connected component of the graph, that the tree has no cycles, that tree edges connect vertices whose BFS levels differ by exactly one, and that every edge in the connected component has endpoints whose BFS levels differ by at most one. All of these checks are simple to perform with KDT's elementwise operators and `SpMV`.

Figure 6 gives Graph500 TEPS scores for both KDT and for a custom C++ code that calls the Combinatorial BLAS engine directly. Both runs are performed on the Hopper machine at NERSC, which is a Cray XE6. Each XE6 node has two twelve-core 2.1 Ghz AMD Opteron processors, connected to the Cray Gemini interconnect. The C++ portions of KDT are compiled with GNU C++ compiler v4.5, and the Python interpreter is version 2.7. We utilized all the cores in each node during

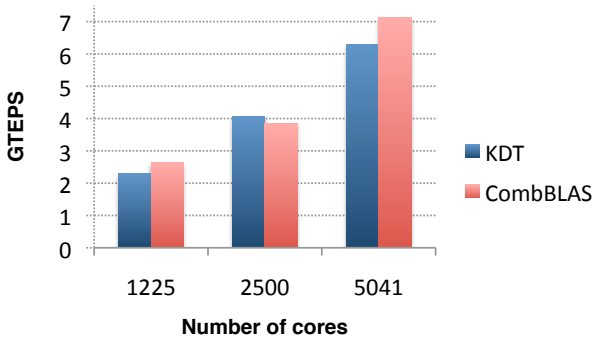

Figure 6: Speed comparison of the KDT and pure CombBLAS implementations of Graph500. BFS was performed on a scale 29 input graph with 500M vertices and 8B edges. The units on the vertical axis are GigaTEPS, or 10^9 traversed edges per second. The small discrepancies between KDT and CombBLAS are largely artifacts of the network partition granted to the job. KDT's overhead is negligible.

Core Count (Machine)	Code	Problem Size		
		Scale 19	Scale 22	Scale 24
4 (Neumann)	PBGL	3.8	2.5	2.1
	KDT	8.9	7.2	6.4
16 (Neumann)	PBGL	8.9	6.3	5.9
	KDT	33.8	27.8	25.1
128 (Carver)	PBGL		25.9	39.4
	KDT		237.5	262.0
256 (Carver)	PBGL		22.4	37.5
	KDT		327.6	473.4

Figure 7: Performance comparison of KDT and PBGL breadth-first search. The reported numbers are in MegaTEPS, or 10^6 traversed edges per second. The graphs are Graph500 RMAT graphs as described in the text.

the experiments. In other words, an experiment on p cores ran on $\lceil p/24 \rceil$ nodes. The two-dimensional parallel BFS algorithm used by Combinatorial BLAS is detailed elsewhere [9].

We see that KDT introduces negligible overhead; its performance is identical to CombBLAS, up to small discrepancies that are artifacts of the network partition granted to the job. The absolute TEPS scores are competitive; the purpose-built application used for the official June 2011 Graph500 submission for NERSC's Hopper has a TEPS rating about 4 times higher (using 8 times more cores), while KDT is reusable for a variety of graph-analytic workflows.

We compare KDT's BFS against a PBGL BFS implementation in two environments. Neumann is a shared memory machine composed of eight quad-core AMD Opteron 8378 processors. It used version 1.47 of the Boost library, Python 2.4.3, and both PBGL and KDT were compiled with GCC 4.1.2. Carver is an IBM iDataPlex system with 400 compute nodes, each node having two quad-core Intel Nehalem processors. Carver used version 1.45 of the Boost library, Python 2.7.1, and both codes were compiled with Intel C++ compiler version 11.1. The test data consists of scale 19 to 24 RMAT graphs. We did not use Hopper in these experiments as PBGL failed to compile on the Cray platform.

The comparison results are presented in Figure 7. We observe that on this example KDT is significantly faster than PBGL both in shared and distributed memory, and that in distributed memory KDT exhibits robust scaling with increasing processor count.

4.2 Betweenness Centrality Betweenness centrality (BC) [14] is a widely accepted importance measure for the vertices of a graph, where a vertex is "important" if it lies on many shortest paths between other vertices. BC is a major kernel of the HPCS Scalable Synthetic Compact Applications graph analysis benchmark [1].

The definition of the betweenness centrality $C_B(v)$ of a vertex v is

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{4.1}$$

where σ_{st} is the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v. Brandes [6] gave a sequential algorithm for BC that runs in $O(ne)$ time on an unweighted graph with n vertices and e edges. This algorithm uses a BFS from each vertex to find the frontiers and all shortest paths from that source, and then backtracks through the frontiers to update a sum of importance values at each vertex.

The quadratic running time of BC is prohibitive for large graphs, so one typically computes an approximate BC by performing BFS only from a sampled subset of vertices [3].

KDT implements both exact and approximate BC by a batched Brandes' algorithm. It constructs a batch of k BFS trees simultaneously by using the `SpGEMM` primitive on $n \times k$ matrices rather than k separate `SpMV` operations. The value of k is chosen based on problem size and available memory. The straightforward KDT code is able to exploit parallelism on all three levels: multiple BFS starts, multiple frontier vertices per BFS,

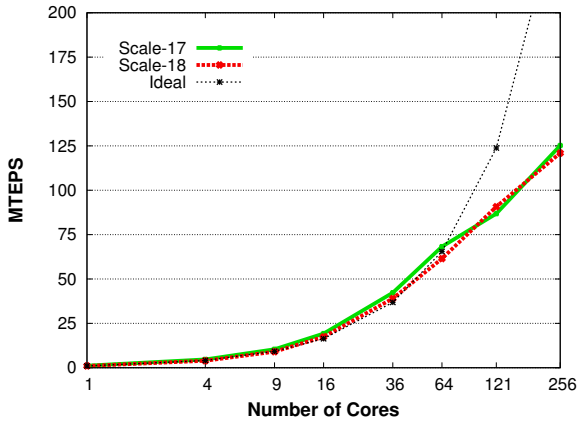

Figure 8: Performance of betweenness centrality in KDT on synthetic power-law graphs (see Section 4.1.2). The units on the vertical axis are MegaTEPS, or 10^6 traversed edges per second. The black line shows ideal linear scaling for the scale 18 graph. The x-axis is in logarithmic scale. Our current backend requires a square number of processors.

and multiple edges per frontier vertex.

Figure 8 shows KDT's performance on calculating BC on RMAT graphs. Our inputs are RMAT matrices with the same parameters and sparsity as described in Graph500 experiments (Section 4.1.2). Since the running time of BC on undirected graphs is quadratic, we ran our experiments on smaller data sets, presenting strong scaling results up to 256 cores. We observe excellent scaling up to 64 cores, but speedup starts to degrade slowly after that. For 256 cores, we see speedup of 118 times compared to a serial run. For all the runs, we used an approximate BC with starting vertices composed of a 3% sample, and a batchsize of 768. This experiment was run on Hopper, utilizing all 24 cores in each node.

4.3 PageRank PageRank [32] computes vertex relevance by modeling the actions of a "random surfer". At each vertex (*i.e.*, web page) the surfer either traverses a randomly-selected outbound edge (*i.e.*, link) of the current vertex, excluding self loops, or the surfer jumps to a randomly-selected vertex in the graph. The probability that the surfer chooses to traverse an outbound edge is controlled by the *damping factor*, d. A typical damping factor in practice is 0.85. The output of the algorithm is the probability of finding the surfer visiting a particular vertex at any moment, which is the stationary distribution of the Markov chain that describes the surfer's moves.

KDT computes PageRank by iterating the Markov chain, beginning by initializing vertex probabilities $P_0(v) = 1/n$ for all vertices v in the graph, where n is the number of vertices and the subscript denotes the iteration number. The algorithm updates the probabilities iteratively by computing

$$P_{k+1}(v) = \frac{1-d}{n} + d \sum_{u \in \mathrm{Adj}^-(v)} \frac{P_k(u)}{|\mathrm{Adj}^+(u)|}, \quad (4.2)$$

where $\mathrm{Adj}^-(u)$ and $\mathrm{Adj}^+(u)$ are the sets of inbound and outbound vertices adjacent to u. Vertices with no outbound edges are treated as if they link to all vertices.

After removing self loops from the graph, KDT evaluates (4.2) simultaneously for all vertices using the `SpMV` primitive. The iteration process stops when the 1-norm of the difference between consecutive iterates drops below a default or, if supplied, user-defined stopping threshold ϵ.

We compare the PageRank implementations which ship with KDT and Pegasus in Figure 9. The dataset is composed of scale 19 and 21 directed RMAT graphs with isolated vertices removed. The scale 19 graph contains 335K vertices and 15.5M edges, the scale 21 graph contains 1.25M vertices and 63.5M edges and the convergence criteria is $\epsilon = 10^{-7}$. The test machine is Neumann (a 32-core shared memory machine, same hardware and software configuration as in Section 4.1.2). We used Pegasus 2.0 running on Hadoop 0.20.204 and Sun JVM 1.6.0_13. We directly compare KDT core counts with maximum MapReduce task counts despite this giving Pegasus an advantage (each task typically shows between 110%-190% CPU utilization). We also observed that mounting the Hadoop Distributed Filesystem in a ramdisk provided Pegasus with a speed boost on the order of 30%. Despite these advantages we still see that KDT is 2 orders of magnitude faster.

Both implementations are fundamentally based on an SpMV operation, but Pegasus performs it via a MapReduce framework. MapReduce allows Pegasus to be able to handle huge graphs that do not fit in RAM. However, the penalty for this ability is the need to continually touch disk for every intermediate operation, parsing and writing intermediate data from/to strings, global sorts, and spawning and killing VMs. Our result illustrates that while MapReduce is useful for tasks that do not fit in memory, it suffers an enormous overhead for ones that do.

A comparison of the two codes also demonstrates KDT's user-friendliness. The Pegasus PageRank implementation is approximately 500 lines long. It is composed of 3 separate MapReduce stages and job management code. The Pegasus algorithm developer must be proficient with the MapReduce paradigm in addition to

Core Count	Task Count	Code	Problem Size Scale 19	Problem Size Scale 21
–	4	Pegasus	2h 35m 10s	6h 06m 10s
4	–	KDT	55s	7m 12s
–	16	Pegasus	33m 09s	4h 40m 08s
16	–	KDT	13s	1m 34s

Figure 9: Performance comparison of KDT and Pegasus PageRank ($\epsilon = 10^{-7}$). The graphs are Graph500 RMAT graphs as described in Section 4.1.2. The machine is Neumann, a 32-core shared memory machine with HDFS mounted in a ramdisk.

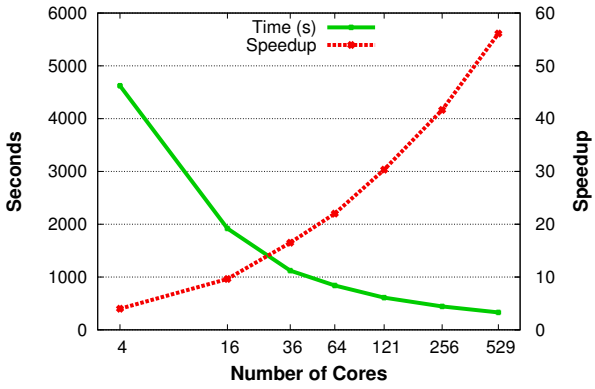

Figure 10: Performance of GaBP in KDT on solving a 500×500 structured mesh, steady-state, 2D heat dissipation problem (250K vertices, 1.25M edges). The algorithm took 400 iterations to converge to a relative norm $\leq 10^{-3}$. The speedup and timings are plotted on separate y-axes, and the x-axis is in logarithmic scale.

the GIM-V primitive. The KDT implementation is 30 lines of Python consisting of input checks and sanitization, initial value generation, and a loop around our SpMV primitive.

4.4 Belief Propagation Belief Propagation (BP) is a so-called "message passing" algorithm for performing inference on graphical models such as Bayesian networks [37]. Graphical models are used extensively in machine learning, where each random variable is represented as a vertex and the conditional dependencies among random variables are represented as edges. BP calculates the approximate marginal distribution for each unobserved vertex, conditional on any observed vertices.

Gaussian Belief Propagation (GaBP) is a version of the BP algorithm in which the underlying distributions are modeled as Gaussian [5]. GaBP can be used to iteratively solve symmetric positive definite systems of linear equations $Ax = b$, and thus is a potential candidate for solving linear systems that arise within KDT. Although BP is applicable to much more general settings (and is not necessarily the method of choice for solving a linear equation system), GaBP is often used as a performance benchmark for BP implementations.

We implemented GaBP in KDT and used it to solve a steady-state thermal problem on an unstructured mesh. The algorithm converged after 11 iterations on the Schmid/thermal2 problem that has 1.2 million vertices and 8.5 million edges [10].

We demonstrate strong scaling using steady-state 2D heat dissipation problems in Figure 10. The $k \times k$ 2D grids yield graphs with k^2 vertices and $5k^2$ edges. We observed linear scaling with increasing problem size and were able to solve a $k = 4000$ problem in 31 minutes on 256 cores. Parallel scaling is sublinear because GaBP is an iterative algorithm with low arithmetic intensity which makes it bandwidth (to RAM) bound. The above experiments were run on Hopper, but we observed similar scaling on the Neumann shared memory machine.

We compared our GaBP implementation with GraphLab's GaBP on our shared memory system. The problem set was composed of structured and unstructured meshes ranging from hundreds of edges to millions. KDT's time to solution compared favorably with GraphLab on problems with more than 10,000 edges.

4.5 Markov Clustering Markov Clustering (MCL) [36] is used in computational biology to discover the members of protein complexes [13, 7], in linguistics to separate the related word clusters of homonyms [12], and to find circles of trust in social network graphs [33, 29]. MCL finds clusters by postulating that a random walk that visits a dense cluster will probably visit many of its vertices before leaving.

The basic algorithm operates on the graph's adjacency matrix. It iterates a sequence of steps called expansion, inflation, normalization and pruning. The expansion step discovers friends-of-friends by raising the matrix to a power (typically 2). Inflation separates low- and high-weight edges by raising the individual matrix elements to a power which can vary from about 2 to 20, higher values producing finer clusters. This has the effect of both strengthening flow inside clusters and weakening it between clusters. The matrix is scaled to be column-stochastic by a normalization step. The pruning step is one key to MCL's efficiency because it preserves sparsity. Our implementation prunes elements which

fall below a threshold though other pruning strategies are possible. These steps are repeated until convergence, then the clusters are identified. The standard, and KDT's default, method is to identify the connected components of the pruned graph as clusters. The KDT Markov clustering method provides sensible defaults for all parameters and options, but allows the user to override them if desired.

4.6 Mini-workflow Example End-to-end graph analysis workflows vary greatly between domains, between problems, and likely even between individual analysts; we do not attempt to describe them here. However, we can identify some smaller mini-workflows as being close enough to real workflows to serve as examples. One mini-workflow, which users say is often applied to power-law graphs resulting from relationship analysis data, has the following steps:

1. Identify the "giant" or largest component
2. Extract the giant component from the graph
3. Find the clusters in the giant component
4. Collapse each cluster into a supervertex
5. Visualize the resulting graph of supervertices

For example, this mini-workflow could analyze Twitter data about politics starting with all people who subscribe to a set of political hash-tags, identifying those people who care strongly about an upcoming election, as evidenced by both sending and receiving political tweets (the giant component), and then clustering them into which candidate they associate with most closely. In KDT this is expressed by the Python code in Figure 2. This mini-workflow illustrates how the KDT methods are designed to work together in sequence. For example, the output of `cluster` (a vector of length equal to the number of vertices in the graph, with each element denoting the cluster in which that vertex resides) is in the same format expected by the `contract` function (which contracts all vertices with the same cluster-ID into a single vertex) and the vertex-partition form of the `nedge` function. The output of this example workflow for a tiny input graph is illustrated in Figure 1.

5 High Level Language Interface

5.1 High Productivity for Graph Analysis KDT targets a demanding environment – domain experts exploring novel very large graphs with hard-to-specify goals. Today this requires knowledge in so many domains that only the most talented, cross-disciplinary, and well-funded groups succeed. KDT aims not only to enable these (non-graph-expert) domain experts to analyze very large graphs quickly but also to accelerate the work of graph-algorithm researchers developing the next generation of algorithms attacking the inherent combinatorial wall of graph analysis.

KDT delivers high productivity to domain experts by limiting the number of new concepts and by providing powerful abstractions for both data and methods. For instance, the `DiGraph` class implements directed graphs for distributed memory, hiding the details of how the directed graph is represented in distributed memory. Similarly, KDT users use the `cluster` method to cluster a graph's vertices by an (initially brief) menu of algorithms. Detailed algorithm-specific options such as the expansion and inflation factors for Markov clustering default to appropriate values for the typical user but enable more knowledgeable users to exercise more control if needed. Those wanting even more control are provided with methods that are too detailed for many domain experts. These include access to well optimized linear algebraic methods and additional graph methods such as `bfsTree` and `normalizeEdgeWeights`

Our experience implementing the primary methods of KDT may illustrate the productivity of this approach. One of us implemented exact betweenness centrality in Python using serial SciPy. Moving that code to run in a distributed parallel manner with KDT required changing the initial definitions of (*e.g.* variable arrays), but much of the core code (*e.g.* multiplying and adding intermediate matrices) did not change. The changes took only 11 hours of programming time for the BC routine itself. The resulting code runs correctly and scales effectively to hundreds of cores. Similarly, after initial explorations to understand the Markov Clustering algorithm and KDT well, an undergraduate student produced our Markov Clustering routine in only six hours.

5.2 Organization of the Fundamental Classes KDT's productivity benefits extend beyond simply providing an opaque set of built-in graph algorithms. The provided set of algorithms also serve as guides for users who want to implement their own graph algorithms based on our extensible primitives.

As Figure 4 illustrates, the `kdt` Python module exposes two types of classes: graph objects and their supporting linear algebraic objects. It includes classes representing directed graphs (`DiGraph`), hypergraphs (`HyGraph`), as well as sparse matrices (`Mat`) and sparse and dense vectors (`Vec`). Computation is performed using a set of pre-defined patterns:

- Matrix-Matrix multiplication (SpGEMM), Matrix-

Vector multiplication (SpMV)

- Element-wise (EWiseApply)
- Querying operations (Count, Reduce, Find)
- Indexing and Assignment (SubsRef, SpAsgn)

These operations are the key to KDT's scalability. Each one is implemented for parallel execution and accepts user-defined callbacks that act similarly to visitors. The pre-defined access patterns allow considerable scalability and account for the bulk of processing time. This allows KDT code to appear serial yet have parallel semantics.

The sparse matrix and vector classes that support the graph classes are exposed to allow complex matrix analysis techniques (*e.g.*, spectral methods). Directed graphs are represented using an $n \times n$ sparse adjacency matrix. Hypergraphs use an $n \times m$ rectangular incidence matrix. Note that bipartite graphs can also be represented with a hypergraph. A graph's edge attributes are represented as the matrix's element values while vertex attributes are stored in vectors of length matching the matrix dimension. KDT's matrices and vectors can be of several types including boolean for connectivity only, floating point, and custom objects.

User-defined callbacks can take several forms. KDT operations accept unary, binary and n-ary operations, predicates, and semiring functions. Each one may be a built-in function or a user-written Python callback or wrapped C routine for speed.

Taken together, these building blocks and finished algorithms provide KDT with a high degree of power and flexibility.

5.3 Semantic graphs Users found that the initial release of KDT lacked support for semantic graphs, *i.e.* graphs whose vertices and edges have types. Semantic graphs are valuable when data is of disparate types (*e.g.* link data about communication via email, Twitter, and Facebook) and considering different types of data together delivers better insight. The KDT semantic graph interface enables on the fly selection of vertices and edges via user-defined callbacks. Computations are only performed on selected vertices and edges. In some situations the graph is very large and the user wants to select most of the graph, in which case materializing the selected graph is wasteful of memory; in other cases the user wants to select only a small portion of the graph, in which case materializing the smaller graph may be more efficient. The KDT semantic graph operations appear to be a dual for SQL's ability to push certain computations onto the database.

```
def onlyEngineers(self):
    return self.position == Engineer

def onlyEmailTwitter(self):
    return self.type == email
        or self.type == Twitter

# the variable G contains the graph
G.addVFilter(onlyEngineers)
G.addEFilter(onlyEmailTwitter)
clus = G.cluster('Markov')
```

Figure 11: Clustering of a filtered semantic graph in KDT. The vertex- and edge-filters consist of predicates which are attached to the graph. They are invoked whenever the graph is traversed.

The subsequent KDT release (v0.2) defines the notion of a *filter*. A filter determines whether or not a particular vertex or edge is included in the computation. Our filter design relies on three basic principles.

1. A user-defined predicate determines whether or not a vertex or edge exists in the filtered graph.
2. Multiple user-defined predicates can be stacked and the filters they define are applied in the order they are added to the graph. Thus, both users and algorithm developers can use filters.
3. All graph operations respect the filter. This ensures that algorithms can be written without taking filters into consideration at all, thus greatly easing their design.

For example, assume that a graph contains link data about communication between employees via email, Twitter, and Facebook, and that a user wants to find clusters in the graph of engineers based on email and Twitter links. This could be implemented with filtered KDT semantic graphs using the code in Figure 11.

We expect the semantic-graph interface to evolve as we continue gathering feedback from KDT users.

6 HPC Computational Engines

6.1 Combinatorial BLAS The Combinatorial BLAS [8] is a proposed standard for combinatorial computational kernels. It is a highly-templated C++ library which serves as the current KDT backend. It offers a small set of linear algebraic kernels that can be used as building blocks for the most common graph-analytic algorithms. Graph abstractions can be built on top of its sparse matrices, taking advantage

of its existing best practices for handling parallelism in sparse linear algebra. Its flexibility comes from the arbitrary operations that it supports. The user, or in this case the KDT implementor, specifies the `add` and `multiply` routines in matrix-matrix and matrix-vector operations, or unary and binary functions for element-wise operations. The main data structures are distributed sparse matrices and vectors, which are distributed in a two-dimensional processor grid for scalability.

We use the publicly available MPI reference implementation of the Combinatorial BLAS as our computational engine. We extended its interface in order to provide further capabilities, such as fully-distributed (to all the processors) sparse vectors, sparse matrix-sparse vector multiplication, and routines akin to MATLAB®'s `sparse` and `find`.

The primary KDT abstractions are different from Combinatorial BLAS abstractions. CombBLAS exposes distributed-memory dense and sparse vectors and sparse matrices and key operations on them, mostly linear algebra, required to implement combinatorial problems. KDT exposes graph abstractions such as directed graphs, and graph operations such as ranking vertices (e.g., betweenness centrality or PageRank), clustering, and finding neighbors within k hops of a set of vertices; the underlying linear algebraic implementation is not immediately visible. This shift in abstractions between the linear-algebra worldview and the graph worldview is one of the primary contributions of KDT. It creates usability for domain experts while retaining performance and customizability.

6.2 Evolution of KDT The design of KDT intentionally separates its user-level language and interface from its computational engine. This allows us to extend KDT easily along at least two axes: an architectural axis, and a capability axis.

On the architectural axis, we intend KDT to map readily to computational engines that provide the functionality of Combinatorial BLAS on different platforms. We and our collaborators are currently working on two such engines: one for manycore shared-address-space architectures, and one for more loosely coupled distributed-computing cloud architectures. We are also contemplating engines that will be able to use more specialized hardware, including GPUs, FPGAs, and massively multithreaded architectures like Cray XMT [21].

On the capability axis, we are extending the set of algorithms and primitives that underlie KDT in various ways. Numerical computational primitives such as linear equation solvers and spectral analysis (computing eigenvalues, singular values, eigenvectors, etc.) are useful in many data analysis settings, and fit naturally into KDT's parallel sparse matrix paradigm. We are also exploring some other classes of graph primitives—for example, the visitor paradigm of the Boost Graph Library and its relatives [34, 22, 17, 4].

In many cases, enhancing KDT's capabilities means interfacing KDT to existing high-performance computational libraries; for example, an upcoming release of KDT is planned to include the numerical PARPACK library [28, 23] in its computational engine, and high-quality high-performance libraries for other numerical computations exist [25, 18, 20].

One of our goals is to use the KDT API as a high-level interface to other existing high-performance graph libraries (such as The MultiThreaded Graph Library [4] and Parallel Boost Graph Library [17]) and representations (such as STINGER [2]). We expect that KDT's high-level language interface will evolve to permit different graph libraries to be used as back ends; we view the current high-level Python specification as a starting point and we are actively soliciting feedback from users and developers to help us guide its evolution.

7 Conclusion

The Knowledge Discovery Toolbox makes truly scalable graph analysis accessible in a high-level language to both domain experts and developers of graph analytics. The two key ingredients are a core set of graph abstractions (and accompanying Python API) providing flexibility and simplicity, and a high-performance computational back end providing scalable performance for graphs in excess of 10 billion edges on HPC clusters. The latest released version of KDT implements the core architecture and a few alternatives for each core operation, which are shown here to enable rapid development of both highly performant graph-analytic workflows and the underlying graph-analytic operations themselves. The performance of KDT approaches that of efficiency-level applications while being reusable for a variety of graph-analytic workflows. In current work, we are extending both KDT's capabilities and the range of hardware and software platforms on which it can be used.

Acknowledgments

We acknowledge support from the Center for Scientific Computing at the CNSI and MRL: a NSF MRSEC (DMR-1121053) and NSF CNS-0960316. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] D. Bader, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS Scalable Synthetic Compact Applications #2. http://graphanalysis.org/benchmark.

[2] D.A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S.C. Poulos. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[3] D.A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. In A. Bonato and F. Chung, editors, *Algorithms and Models for the Web-Graph*, volume 4863 of *Lecture Notes in Computer Science*, pages 124–137. Springer Berlin/Heidelberg, 2007.

[4] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *Proc. Workshop on Multithreaded Architectures and Applications*. IEEE Press, 2007.

[5] D. Bickson. Gaussian Belief Propagation: Theory and Application. *CoRR*, abs/0811.2518, 2008.

[6] U. Brandes. A Faster Algorithm for Betweenness Centrality. *J. Math. Sociol.*, 25(2):163–177, 2001.

[7] S. Brohée and J. van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC Bioinformatics*, 7:488, 2006.

[8] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[9] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Supercomputing*, 2011.

[10] T.A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, to appear, 2011.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating System Design and Implementation*, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association.

[12] B. Dorow. *A Graph Model for Words and their Meanings*. PhD thesis, Universität Stuttgart, 2006.

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

[14] L.C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

[15] A.V. Goldberg and R.F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX05)*, pages 26–40, 2005.

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

[17] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *Proc. Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'05)*, 2005.

[18] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.*, 31:397–423, September 2005.

[19] U. Kang, C.E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[20] G. Karypis, K. Schloegel, and V. Kumar. ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Technical report, Dept. of Computer Science, University of Minnesota, 1997.

[21] P. Konecny. Introducing the Cray XMT. *Cray User Group meeting (CUG)*, 2007.

[22] L.Q. Lee, A. Lumsdaine, and J.G. Siek. The Boost Graph Library: User Guide and Reference Manual, 2002. www.osl.iu.edu/publications/Year/2002.complete.php.

[23] R.B. Lehoucq, D.C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[24] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *PKDD*, pages 133–145. Springer, 2005.

[25] X.S. Li, J.W. Demmel, J.R. Gilbert, L. Grigori, and M. Shao. *SuperLU Users' Guide*, 2010.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J.M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[27] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[28] K.J. Maschho and D.C. Sorensen. A portable implementation of ARPACK for distributed memory parallel architectures. In *Proceedings of the Copper Mountain Conference on Iterative Methods*, pages 9–13, April 1996.

[29] J. McPherson, K.-L. Ma, and M. Ogawa. Discovering Parametric Clusters in Social Small-World Graphs. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 1231–1238, New York, NY, USA, 2005. ACM.

[30] H. Meuer, E. Strohmaier, J.J. Dongarra, and H.D.

Simon. Top500 supercomputer sites. In *Proc. SC2001*, pages 10–16, 2001. http://www.top500.org.

[31] M. Najork and J. L. Wiener. Breadth-First Search Crawling Yields High-Quality Pages. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 114–118, New York, NY, USA, 2001. ACM.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[33] A. Petróczi, Tamás Nepusz, and Fülöp Bazsó. Measuring tie-strength in virtual social networks. *CONNECTIONS - the official journal of the International Network for Social Network Analysis*, 27(2):49–57, 2006.

[34] J. Siek, A. Lumsdaine, and L.Q. Lee. Boost Graph Library, 2001. http://www.boost.org/libs/graph/doc/index.html.

[35] Titan Informatics Toolkit. http://titan.sandia.gov.

[36] S. van Dongen. Graph Clustering via a Discrete Uncoupling Process. *SIAM J. Matrix Anal. Appl*, 30(1):121–141, 2008.

[37] J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding Belief Propagation and its Generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.