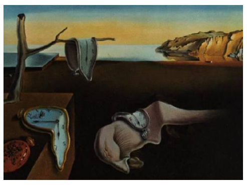


Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Analysis and Visualization of Network Data using JUNG

Joshua O'Madadhain[*] University of California, Irvine

Danyel Fisher[*] Microsoft Research

Padhraic Smyth University of California, Irvine

Scott White University of California, Irvine

Yan-Biao Boey Mercury Interactive

Abstract

The **JUNG** (Java Universal Network/Graph) Framework is a free, open-source software library that provides a common and extendible language for the manipulation, analysis, and visualization of data that can be represented as a graph or network. It is written in the Java programming language, allowing **JUNG**-based applications to make use of the extensive built-in capabilities of the Java Application Programming Interface (API), as well as those of other existing third-party Java libraries. We describe the design, and some details of the implementation, of the **JUNG** architecture, and provide illustrative examples of its use.

Keywords: network, graph, representation, visualization, data analysis, framework, Java, software library, open-source.

1. Introduction

A network data set typically consists of a set of entities and known relationships among these entities. For example, a social network data set could consist of a list of individuals and a list of pairwise binary relations indicating those pairs of individuals that are known to be friends. It is often convenient to formally represent such data as a graph, with vertices representing entities and edges representing their relationships. Examples of networks of broad interest include bibliographic citation networks (papers citing other papers), biological networks (e.g., proteins and their interactions), telecommunication networks, and the Web.

Network data sets can also include additional information about both the entities and the rela-

[*]The first two authors listed on the paper contributed equally to the development of the JUNG software package.

tionships. A co-authorship network, for example, could be defined on authors and co-author relationships. Additional information about each author could include the author's name, institutional affiliations, and areas of interest. Similarly, the documents underlying the co-author relationships have keywords, text content, and citations to other papers. Furthermore, authors in such a network may also be related in other ways (advisory relationships, memberships on conference committees, friendships, etc.). These kinds of multilayered relationships are not necessarily amenable to the same kinds of well-known data analytic techniques that apply to "flat" or non-relational data.

The study of such network data sets is becoming increasingly common across a variety of disciplines, including statistics, social science, computer science and engineering, physics, and biology. In turn this has led to a diverse set of data analysis techniques being developed. Among these techniques are methods for graph visualization and drawing (Battista, Eades, Tamassia, and Tollis (1999)), physically-motivated models for characterizing large-scale properties of networks (Watts and Strogatz (1998), Barabasi and Albert (1999), Newman (2000)), Markov and eigenvector models for Web page modeling (Brin and Page (1998), Kleinberg (1999)), and quantitative methodologies for analyzing social networks (Wasserman and Faust (1994)) including relatively sophisticated statistical models (Hoff, Raftery, and Handcock (2002), Butts (2003), Huisman and Snijders (2003)).

Across all of these techniques many of the same basic operations are routinely performed, such as aggregating over vertices and edges to compute summary statistics or computing shortest paths between vertices. Performing these tasks can be significantly more complicated than traditional analysis tasks for "flat file" multivariate data. This is particularly true for the types of large sparse networks that are common in many applications, with large numbers of vertices but relatively few edges per vertex on average. Data structures that are specifically designed for sparse graphs can be significantly more computationally efficient and more flexible to work with compared to using "non-sparse" multivariate alternatives.

The challenges of working with network data motivate the creation and development of flexible software environments that are designed specifically for such data. In this paper we describe the **JUNG** Framework, an open-source Java-based software library that has been developed specifically as a common and extendible language for the manipulation, analysis, and visualization of data that can be represented as a graph or network.

The major features of **JUNG** include the following:

- Support for a variety of representations of entities and their relations, including directed and undirected graphs, multi-modal graphs (graphs which contain more than one type of vertex or edge), graphs with parallel edges (also known as multigraphs), and hypergraphs (which contain hyperedges, each of which may connect any number of vertices).
- Mechanisms for annotating graphs, entities, and relations with metadata. These capabilities facilitate the creation of analytic tools for complex data sets that can examine the relations between entities, as well as the metadata attached to each entity and relation.
- Implementations of a number of algorithms from graph theory, exploratory data analysis, social network analysis, and machine learning. These include routines for clustering, decomposition, optimization, random graph generation, statistical analysis, and calculation of network distances, flows, and ranking measures (centrality, PageRank, HITS, etc.)

- A visualization framework that makes it easy to construct tools for the interactive exploration of network data. Users can choose among the provided layout and rendering algorithms, or use the framework to create their own custom algorithms.
- Filtering mechanisms which extract subsets of a network; this allows users to focus their attention, or their algorithms, on specific portions of a network.

These capabilities make **JUNG** a good platform for exploratory data analysis on relational data sets.

JUNG is a framework on which applications and tools for manipulating graph and network data can be built. It can be used in simple snippets of code to test ideas, or to aid in the development of a sophisticated tool with a graphic user interface. **JUNG** is not itself a stand-alone tool, but rather a library that can be used to support the construction of specific tools. Building a tool that uses **JUNG** requires some knowledge of Java programming. The **JUNG** distribution does include samples of small applications that use **JUNG** to accomplish certain tasks, but they are intended to be examples of how one might use **JUNG**, not tools in their own right.

As a library, **JUNG** can be used both to build network-oriented tools, and to provide network capabilities to existing systems. For example, **JUNG** has been used to build Netsight, which is a stand-alone application that will be discussed in Section 11, and has also been used to add network visualizations to the code-analysis tool Augur (Froehlich and Dourish (2004)).

One advantage of using a toolkit, rather than an existing tool, for network analysis is that it can be scripted in a very general way. This flexibility has made it possible for JUNG to be used as part of a web-client applet that dynamically reads in and visualizes networks from a database (Fisher and Dourish (2004)), to repeat a set of operations on several thousand networks (Smith and Fisher (2005)), and to be addressed by scripts in the network visualization tool GUESS (Adar and Feinberg (2005)).

JUNG is freely provided under the BSD open-source license, which allows anyone to create derived works from **JUNG**, as long as they acknowledge **JUNG**'s contribution to their work. The open-source nature of the project makes it easier for users to write their own code, and to understand how **JUNG** itself works; as a result, members of **JUNG**'s user community have been able to contribute a number of extensions, algorithms and fixes.

We will use a data set derived from the CiteSeer digital library (Lawrence, Giles, and Bollacker (1999)) as a motivating example throughout this paper. CiteSeer is a repository of more than 500,000 papers, written by approximately 100,000 different authors, primarily from computer science. Each of these papers has been parsed to recover the text of the paper, information about its authors (such as names and institutional affiliations), and its references to other papers. This data can be interpreted to produce several different kinds of networks: a bipartite network of papers and authors, a citation network of papers, a co-authorship network of authors, a co-citation network of papers or of authors, and so on. Networks of this size and complexity present several challenges to the data analyst.

The remainder of this paper is organized in the following sections.

In Section 2, we briefly review some of the concepts and terminology of graph theory and object-oriented programming.

Sections 3 and 4 provide background on the basic principles of **JUNG**'s design and operation. In Section 3, we discuss some of the reasoning behind the design of the **JUNG** architecture; in

Section 4, we describe the fundamental properties and operations of **JUNG**'s graph, vertex, and edge objects.

Sections 5, 6, 7, and 8 describe some of **JUNG**'s capabilities for representing and processing data. Section 5 outlines methods for associating metadata with each **JUNG** element; Section 6, describes **JUNG**'s filtering mechanisms, which create subsets of existing networks; Section 7 outlines **JUNG**'s support for handling separate computational processes that all refer to the same (dynamically changing) network; and Section 8 lists some of **JUNG**'s algorithms for network analysis.

Sections 9 and 10 describe **JUNG**'s input and output systems. Section 9 outlines **JUNG**'s architecture for creating network visualizations, and Section 10 briefly describes the ways in which graphs may be created from and saved as persistent data storage.

Sections 11 and 12 discuss applications of **JUNG**. Section 11 presents case studies of how **JUNG** has been used, and Section 12 compares **JUNG** to other tools and libraries for network analysis and visualization.

Finally, in Section 13, we mention some plans for **JUNG**'s future development.

2. Terminology and Notation

This section briefly defines some of the terms that are used in this paper, for those who may not be familiar with graph theory or with object-oriented programming in Java.

2.1. Graph Theory

Graphs (also known as *networks*) consist of a set of *vertices*, V, and a set of *edges*, E; the number of vertices is denoted by $|V|$ and the number of edges by $|E|$. Vertices (also known as *nodes*) represent entities, and edges (also known as *arcs*, *links*, or *ties*), which connect vertices, represent relationships or events which involve the entities that the vertices represent. The number of edges incident to a vertex is called the *degree* of that vertex. Graphs in which each edge has an associated numeric value (such as the number of co-authored papers) are called *weighted* or *valued* graphs. One common graph subtype is a *k-partite graph* (called a *bipartite graph* when $k = 2$), in which the vertices are partitioned into k disjoint subsets, and each edge connects vertices in distinct partitions.

Most graphs contain edges that each connect exactly two vertices; unless otherwise specified, all graphs in this paper have this property. (Graphs in which edges can connect any number of vertices are called *hypergraphs*, and their edges are called *hyperedges*.) An edge which has a defined source and destination (such as one representing "A has cited B") are called *directed* edges; an edge which does not (such as one representing "A and B have co-authored a paper" are called *undirected* edges. Graphs which contain both directed and undirected edges are called *mixed-type* graphs. Two edges are said to be *parallel* if they connect the same set of vertices and have the same direction/ordering.

A graph is said to be *connected* if each vertex is reachable from each other vertex; many algorithms (such as centrality algorithms) are only well-defined on connected graphs. We define the *distance* between two vertices to be the length of the shortest path (on the underlying unweighted graph) that connects them; in other contexts, the distance may refer to the shortest weighted path.

A network may contain entities of different types, or with different roles; it may also include different types of relationships or events. These roles and interaction types are collectively referred to as *modes*. A network which has one type of entity and one type of relationship is called a *single-modal network*; if the network has more than one type of entity, relationship, or both, it is called a *multi-modal* network.

2.2. Object-Oriented Programming and Java

Java is an object-oriented programming language. This generally means that programming involves (a) defining object types and their capabilities, and (b) constructing objects and using their capabilities in aid of the desired tasks.

Object types are defined by *interfaces* and *classes* (which may *implement* one or more interfaces, and may *extend* (inherit behavior from, or be a *subclass* of) a single other class). A particular object is said to be an *instance* of the types that it implements and extends. Java defines a class called `Object` which is a superclass of all Java classes (including, implicitly, any user-defined class). The specific behaviors and capabilities of a class are defined by the implementation of that class's *methods*; the types and ordering of a method's arguments define that method's *signature*. Classes may be declared to be *abstract*, in which case they need not supply implementations of each method that they declare, and an instance of the class cannot be created; this can be useful for providing implementations that are general enough to apply to most anticipated extensions of such a class. Objects may contain references to other objects.

API (Application Programming Interface) is a common term for a software library, especially one whose design philosophy emphasizes backwards compatibility as the library evolves. Both the standard Java libraries and **JUNG** are APIs.

There are a few different Java APIs that can be used to create graphic user interfaces; two of the most popular are Swing (Sun Microsystems (2004)) and SWT (Eclipse Foundation (2001)).

3. Design Principles

In this section, we discuss the key principles behind the design of **JUNG**.

3.1. Abstraction

JUNG makes use of Java interfaces, abstract classes, and implementation classes in its type definitions. There are a few reasons that **JUNG** uses combinations of these layers of abstraction.

First, it separates the specifications from their implementations; this makes **JUNG** more useful as a general language for handling graphs as data. Thus, for example, the `ArchetypeGraph` interface specifies the capabilities of all graphs, but does not constrain how those capabilities are provided.

Second, we combine this notion of abstraction with the object-oriented idea of inheritance. For instance, we define `Graph`, the interface for all graphs whose edges connect exactly two vertices, as a sub-interface of `ArchetypeGraph`, which means that it has all the properties of `ArchetypeGraph`, but with a few additional properties that are specific to such graphs; this reflects the actual relationship between "archetype graphs" and graphs. Similarly, the abstract class `AbstractArchetypeGraph` provides implementations of the methods of `ArchetypeGraph` that aren't specific to any particular variety of graph.

Third, it is advantageous to design method signatures, where possible, such that users cannot supply arguments of the wrong type; this allows users to discover such errors when they compile their code, rather than when they run it. At the same time, methods should generally not be so constrained that they can only accept arguments which use a particular internal implementation. We achieve both of these aims by specifying method signatures in terms of interfaces (which only specify capability) rather than classes. Thus, for example, **JUNG**'s methods generally operate on `Graph`s, rather than `SparseGraph`s; this allows users to use these methods with their own `Graph` implementations, if they choose, as well as with any implementations of `Graph` that **JUNG** may provide in future.

3.2. Performance

Traditionally, network and graph data sets have often been described mathematically as matrices. Matrices are commonly implemented as two-dimensional arrays; this representation facilitates fast retrieval of the edge(s), if any, that connect(s) two specified vertices (which operation is called `findEdge` in **JUNG**). (For example, in the context of the CiteSeer co-authorship network, one could use `a.findEdge(b)` to retrieve the edge which maintains a list of the papers for which researchers `a` and `b` have appeared as co-authors.) However, this representation is generally not feasible for large-scale networks. First, it requires $O(|V|^2)$ space. Thus, a 2D array representation of the CiteSeer network, which has about 100,000 authors, would require 10 billion entries to represent the possible connections; this far exceeds the memory capacity of any currently available desktop computer. Second, existing algorithms for network analysis which operate on matrices do not scale well to matrices of this size; many matrix-based network analysis algorithms involve matrix multiplication, or matrix inversion, which generally require $O(|V|^3)$ time on 2D arrays. In addition, this representation is problematic for dynamic networks (those whose vertex set may grow larger or smaller) and for networks with parallel edges, and does not provide a natural way to represent metadata for entities. Finally, large-scale networks are almost invariably very sparse, so almost all of the space in a 2D array representing such a network is wasted on representing absent links.

A common alternative representation for sparse graphs and networks is the adjacency list representation, in which each vertex maintains a list of incident edges (or adjacent vertices); this requires $O(|V| + |E|)$ space. In the case of the CiteSeer network, this represents a savings in space of a factor of approximately 10^4. Unlike the 2D array representation, this representation does not permit an efficient implementation of `findEdge`; nevertheless, **JUNG** provides an implementation of this representation, for circumstances in which saving space is paramount.

Most of the current **JUNG** vertex implementations employ a variant of the adjacency list representation, which we term an adjacency map representation: each vertex maintains a map from each adjacent vertex to the connecting edge (or connecting edge set, in the case of graphs that permit parallel edges). (Separate maps are maintained, if appropriate, for incoming directed edges, outgoing directed edges, and undirected edges.) This uses slightly more memory than the adjacency list representation, but makes `findEdge` approximately as fast as the corresponding operation on the 2D array representation. This representation makes **JUNG**'s data structures and algorithms, in general, well-suited for use on large sparse networks (O'Madadhain, Smyth, and Adamic (2005), White and Smyth (2003)).

3.3. Predicates

A predicate is an expression (for example, "`v.degree() > 3`") that, when evaluated on a specified argument (in this case, `v`), returns either "true" or "false". Arguments for which a predicate returns "true" are said to pass this predicate, and those for which the predicate returns "false" are said to fail it. Some predicates, such as logical operators, act on other predicates (such as an `AND` predicate, which its argument `a` passes if and only if `a` passes each of its constituent predicates).

Graphs can impose constraints on the vertices or edges that may be added to them, such as "only directed edges", or "no parallel edges". These constraints, as well as user-defined criteria, can be expressed using predicates. For example, in a statistical analysis of the CiteSeer co-authorship network, a user could use predicates to interactively define two sets of edges (e.g., one consisting of papers published before the year 2000, and the other for papers from 2000 onwards), and then test a hypothesis that the two edge sets came from different edge distributions.

Predicates can also serve as flexible specifications for filters; for example, a renderer can be told to only draw edges that pass a specified predicate, such as "`weight(e) > x`", and then vary the value of `x` according to the state of a visual interface component (such as a slider). (For an example of a filtering interface, see Figure 9.)

These two ideas (constraining and filtering) are combined to form **JUNG**'s support for specific constrained graph types. For example, **JUNG** provides support for k-partite graphs via the `KPartiteSparseGraph` class. Instances of this class are created with a collection of vertex predicates; **JUNG** then automatically creates an edge predicate that only passes edges which connect vertices from distinct subsets. In addition, these vertex predicates are used as subset specifications; **JUNG** provides utility methods that return all the vertices (or edges) that pass a specified predicate.

3.4. Using JUNG With Other Libraries

Just as users can write Java programs that use **JUNG** to provide methods for manipulating network data, **JUNG** uses other Java libraries (which are also platform-independent), to perform certain functions and thus reduce the duplication of effort.

Commons Collections (Apache Jakarta Project (2004)) is a library that enhances the basic Java API for collections of objects (for instance, by providing basic set-theoretic operations). **JUNG** makes extensive use of this library; in particular, Commons Collections provides the `Predicate` interface, which **JUNG** uses to implement support for predicates, subsets, and constraints.

Colt (CERN (2004)) is a set of libraries for high-performance scientific and technical computing. **JUNG** provides methods to convert **JUNG** graphs to Colt matrices, and vice versa; this allows users of **JUNG** to apply Colt's extensive library of linear and matrix algebra routines to **JUNG** graphs. **JUNG** also makes occasional use of some of Colt's matrix algebra functions.

Xerces (Apache XML Project (2004)) is a library for parsing XML, which is used to implement the GraphML (Brandes, Eiglsperger, Kaufmann, and Lerner (2004)) input/output capabilities. These capabilities are discussed in Section 10, Input and Output.

It is also possible to use **JUNG** in conjunction with code written in other languages: such code can make use of **JUNG**, extend **JUNG**'s capabilities, or both. The Java Native Interface (Sun Microsystems (2003)) allows Java code to access, and be accessed by, so-called "native" code (that is, code which is written for a particular programming platform, such as Windows or Linux). The **SJava** package (Lang (2004)) provides a more specialized way of calling R and S code from Java and vice versa.

4. Graphs, Vertices, and Edges

4.1. Basic Properties and Operations

Graphs, vertices, and edges each have several properties that can be extracted, and operations that they can perform (or have performed upon them). JUNG's interfaces, outlined below, define these properties and operations. Depending on the specific type of graph, vertex, or edge, and on the implementation used, a given object may have other available properties and/or operations. Each of these basic types includes operations for creating a copy of itself; vertices and edges also include operations for identifying such a copy in a specified graph. (Copying will be discussed in more detail in Section 4.4.)

By definition, **JUNG** vertices and edges can be elements of no more than one graph; this allows users to ask, for example, for a vertex's neighboring vertices without requiring them to specify the graph as part of the query. This implies that each vertex and edge should be able to identify the graph of which it is an element; this information can be of use, for example, in writing a method for determining the length of the shortest path between two specified vertices (where the graph itself is not explicitly specified).

Graphs

`ArchetypeGraph` defines a graph to be a container of vertices and edges, with several methods for accessing and modifying these sets, for defining vertex and edge constraints, and for specifying listeners. (Listeners are discussed in Section 7, Event Handling.)

The `Hypergraph` and `Graph` interfaces extend the `ArchetypeGraph` interface, and are specialized for hypergraphs and graphs whose edges connect exactly two vertices, respectively. These interfaces specify method signatures for adding vertices and edges; these methods are defined here rather than in `ArchetypeGraph` so as to allow for compile-time checks of vertex and edge type (as discussed in Section 3.1, Abstraction). `Graph` in turn has subinterfaces for graphs that have only directed edges, only undirected edges, or that are k-partite. The `Sparse` graph types provide implementations of the `Graph` methods that are suitable to a sparse graph. The graph type hierarchy is shown in Figure 1.

Vertices

The `ArchetypeVertex` interface defines a vertex as an entity, which resides in a specified graph, with incident edge and neighboring vertex sets; it also defines various convenience and utility methods (such as `findEdge`, which was discussed in Section 3.2).

The `Vertex` interface extends the `ArchetypeVertex` interface, and specifies additional method

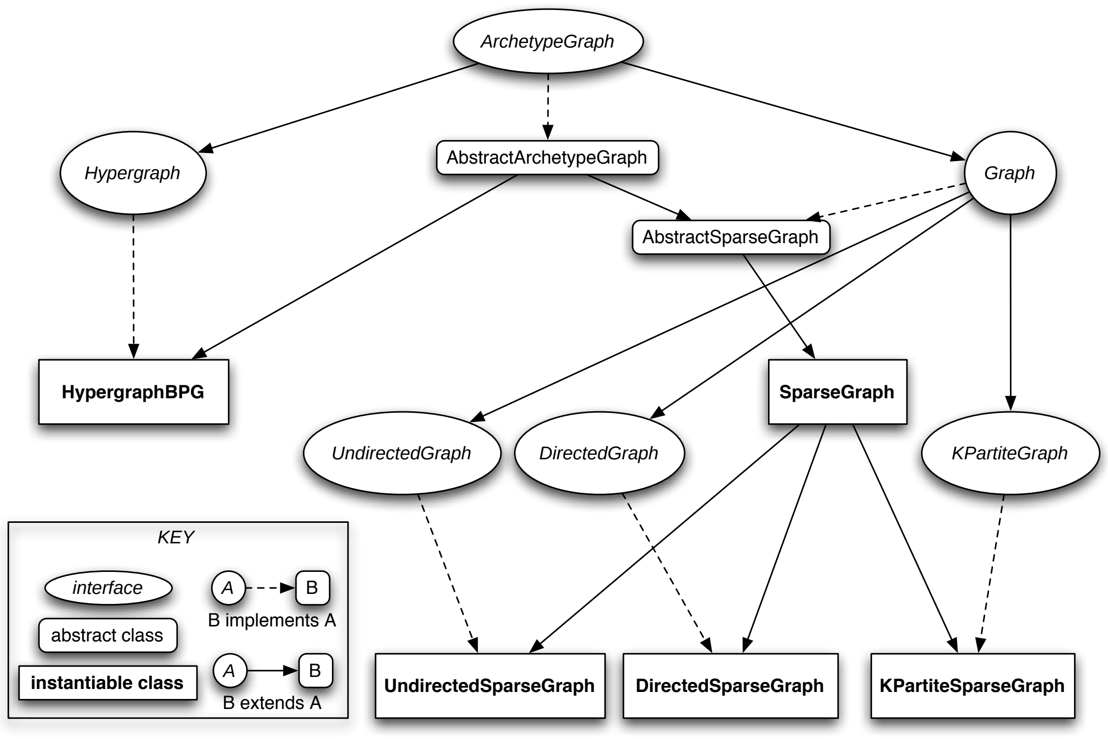


Figure 1: The type hierarchy for graphs

signatures for vertices of graphs whose edges connect exactly two vertices; these include methods for accessing a vertex's successors and predecessors, or its outgoing and incoming edges. In such graphs, a vertex `v` that is connected to another vertex `w` via an undirected edge `e` is both `w`'s successor and its predecessor (and vice versa), and `e` is considered to be both an incoming and an outgoing edge of `v` and of `w`.

The `Hypervertex` interface also extends the `ArchetypeVertex` interface; it serves to identify a vertex that may be added to a `Hypergraph`.

The `AbstractSparseVertex` abstract class implements some of the basic `Vertex` methods, and is in turn extended by several different vertex classes, each of which is designed for a different combination of graph properties (such as whether the graph accepts only directed edges, only undirected edges, or both, and whether it accepts parallel edges). This multiplicity of types permits more efficient `Vertex` implementations for more constrained graph types (such as directed graphs which do not permit parallel edges). The vertex type hierarchy is shown in Figure 2. In some cases, it can be useful to treat a directed graph `g` as an undirected graph (for example, when deciding how to draw a picture of it). Since **JUNG** considers successors and predecessors to be special cases of neighbors, and outgoing and incoming edges to be special cases of incident edges, one can implicitly look at the underlying undirected graph of `g` by accessing the neighbor and incident edge sets for each vertex.

Edges

The `ArchetypeEdge` interface defines an edge to be an entity, which resides in a specified

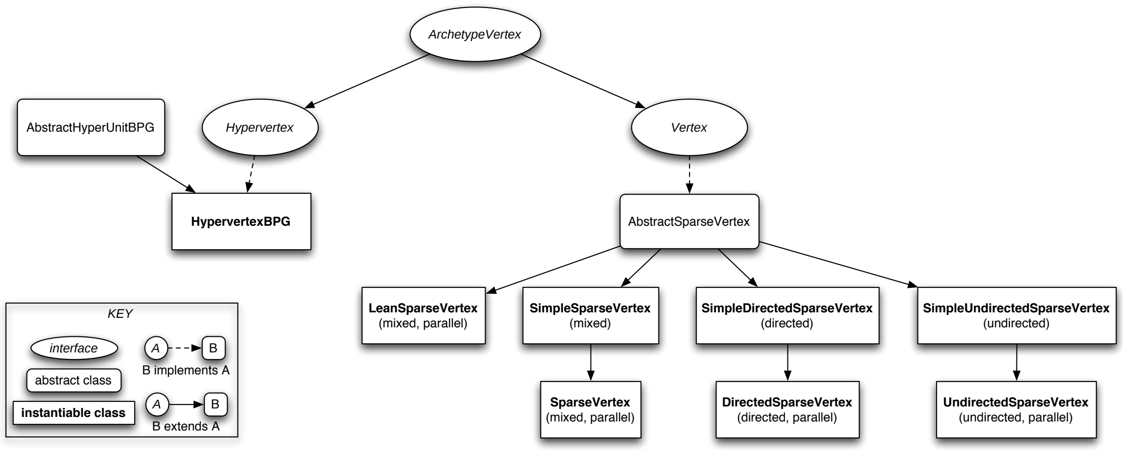

ArchetypeVertex
AbstractHyperUnitBPG
Hypervertex
Vertex
HypervertexBPG
AbstractSparseVertex
LeanSparseVertex (mixed, parallel)
SimpleSparseVertex (mixed)
SimpleDirectedSparseVertex (directed)
SimpleUndirectedSparseVertex (undirected)
SparseVertex (mixed, parallel)
DirectedSparseVertex (directed, parallel)
UndirectedSparseVertex (undirected, parallel)
KEY
interface
abstract class
instantiable class
B implements A
B extends A

Figure 2: The type hierarchy for vertices

graph, with an incident vertex set; it also defines various convenience and utility methods (such as one which returns `true` if a specified vertex is incident to this edge).

The `Edge` interface extends the `ArchetypeEdge` interface for edges which connect exactly two vertices; it is in turn extended by interfaces which define additional operations for directed and undirected edges.

The `Hyperedge` interface also extends the `ArchetypeEdge` interface, and provides a method signature for adding a vertex to an existing hyperedge.

The `Sparse` implementations of `Edge` correspond to the `Sparse` implementations of `Vertex` and `Graph`. The edge type hierarchy is shown in Figure 3.

4.2. Creating, Adding, and Removing

Creating a graph may be done in three ways. First, one can call the constructor for the desired type of graph, as in the following example:

```
DirectedGraph g = new DirectedSparseGraph();
```

Second, one can create a graph by reading it in from a file. Currently, **JUNG** can read and write simple Pajek (Batagelj and Mrvar (2004)) and GraphML (Brandes *et al.* (2004)) files; this is discussed in more detail in Section 10.

Third, one can generate a graph algorithmically, either with a user-defined method (e.g., one that generates a graph based on a specified stochastic model), or with one of the classes that **JUNG** provides for creating random graphs.

Once a graph has been created using any of these methods, vertices may be created and added to this graph:

```
Vertex v1 = g.addVertex(new DirectedSparseVertex());
Vertex v2 = g.addVertex(new DirectedSparseVertex());
```

and once vertices exist, they may be connected by edges:

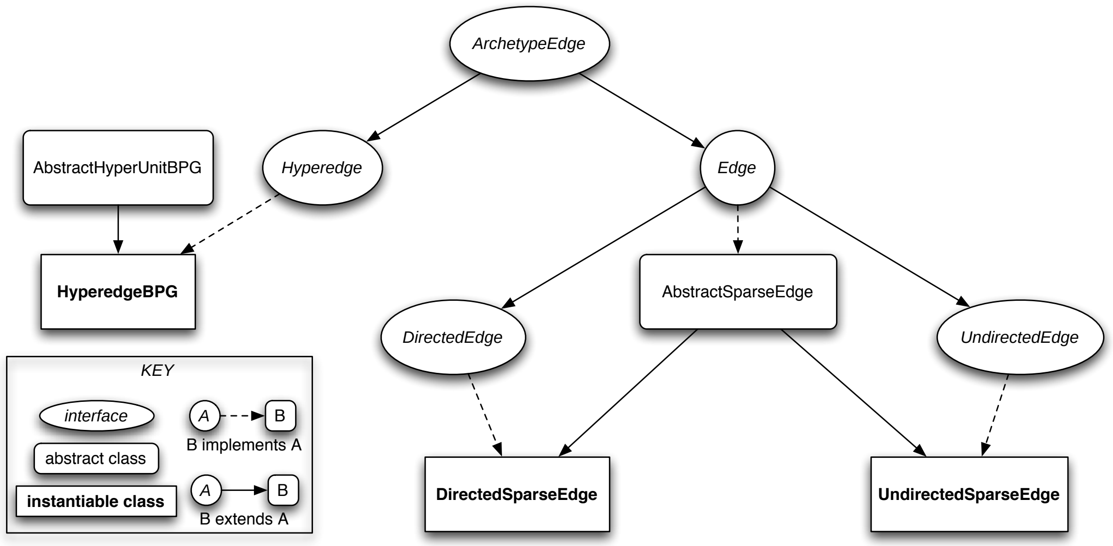


Figure 3: The type hierarchy for edges

```
Edge e = new DirectedSparseEdge(v1, v2);
g.addEdge(e);
```

These operations may be used to update the CiteSeer co-authorship network as new authors appear (new vertices), and new papers are written (new edges).

Note that creating vertices/edges and adding them to a graph are actually two different operations, which can be combined into a single line of code (as in the vertex creation/addition example above). The two-stage nature of this process makes it possible to create "orphaned" vertices/edges that are not part of a graph. The behavior of the **JUNG** edge and vertex methods is generally unspecified on orphaned vertices/edges.

This distinction (between creation and addition) is useful, however, because it allows us to decorate vertices/edges (that is, attach data to them) before adding them to a graph:

```
Vertex v1 = new DirectedSparseVertex();
setProfession(v1, "Professor");
g.addVertex(v1);
```

In particular, if `g` contains a vertex constraint that requires that each vertex have a defined "profession", or if `g` is maintaining a list of predicate-specified subsets (one for each profession), then the vertex must be so decorated before it is added to the graph.

Removing vertices and edges from a graph can be useful in several contexts, such as the identification of duplicate or otherwise invalid vertices/edges (the raw CiteSeer data set contains a number of duplicate papers and authors); the creation of subgraphs; and the operation of an algorithm that is testing the effects of (temporary) changes to the graph. This can be done as follows:

```
g.removeEdge(e);
```

```
    g.removeVertex(v1);
```

Removing an edge from a graph will not affect any other part of the graph. Removing a vertex from a graph may cause the edges that are incident to that vertex to be removed if these edges would otherwise become ill-formed. (An ill-formed edge is one that is incident to the wrong number of vertices. In graphs where edges are defined to connect exactly two vertices, removing a vertex will result in the removal of all of its incident edges.)

Removing an element from a graph does not free the memory used by that object; in fact, an element can be removed from a graph and then re-insert it in that graph or in a different graph). As with all Java programs, the Java garbage collector is responsible for freeing the memory for an object once it is no longer being used. Removing an element from a graph also does not remove it from any **JUNG**-provided user data repositories (discussed in Section 5.2); users are responsible for updating the user data as necessary.

4.3. Constraints

The following constraints apply to all **JUNG** edges, vertices, and graphs:

- A vertex/edge may only be in one graph at a time.
- A vertex/edge may only be added to a given graph once.
- An edge may not be created incident to "orphaned" vertices.
- An edge may not be created which joins vertices in different graphs.
- The directionality of an edge must match that of the vertices that it is connecting, and that of the graph to which it is being added. (Thus, for example, a `DirectedSparseEdge` may not be added to an `UndirectedGraph`.)

As outlined in Section 3.3, Predicates, vertex and edge constraints are defined using the `Predicate` interface, and may be accessed (and edited) via the `getVertexConstraints` and `getEdgeConstraints` methods provided by `ArchetypeGraph`:

```
    Collection edge_constraints = g.getEdgeConstraints();
    // disallow parallel edges
    edge_constraints.add(Graph.NOT_PARALLEL_EDGE);
    // disallow self-loops (edges that connect a vertex to itself)
    edge_constraints.add(new NotPredicate(SelfLoopPredicate.getInstance());
```

Constraints may be freely edited for empty graphs, but cannot be added to non-empty graphs (this avoids consistency problems, since constraints are only checked when vertices and edges are added to a graph).

Not all predicates are appropriate for use as constraints; for example, `ReciprocatedDirectedEdgePredicate` only passes directed edges `<u,v>` for which there also exists an antiparallel edge `<v,u>`; since edges are added to a graph one at a time, no edge can pass this predicate when it is checked as a constraint. The documentation for the predicates that **JUNG** provides generally specify whether they can be used as constraints.

4.4. Copying and Equivalency

It can be useful to represent a single entity (or relationship) as two or more vertices (or edges) in different networks. It is also sometimes desirable or useful to make a copy of a graph, vertex, or edge; for example, this permits users to experiment with mutations of the graph, to create subgraphs, or to observe in parallel different scenarios for the evolution of a graph. For example, one could create a subgraph of the full CiteSeer co-authorship network in which each vertex represents an author who has authored at least 50 papers, and then analyze the connectivity of this "prolific author" subgraph. In such cases, it can be useful to ask the question "what is the vertex `v'` in graph `g'` (the copied/derived graph) which is the equivalent to vertex `v` in graph `g` (the original graph)?"

When a vertex, edge, or graph is copied, a new object of the same type is created, and the contents of the object's **JUNG** user data repository (about which more in Section 5.2) are copied from the original object to the copy. If the object being copied is a graph, then all of its vertices and edges are also copied into the new graph. If the object is a vertex or edge, then the copied object is added to the specified target graph, and an equivalence relation is created between the original vertex/edge (and, implicitly, any vertices/edges to which the original vertex is equivalent) and the copy.

The following code creates a graph, creates two vertices and an edge and adds them to this graph, then copies each vertex and edge from the original graph to a new target graph.

```
Graph original = new DirectedSparseGraph();
Vertex v1 = original.addVertex(new DirectedSparseVertex());
Vertex v2 = original.addVertex(new DirectedSparseVertex());
Edge e = original.addEdge(new DirectedSparseEdge(v1, v2));

Graph target = new DirectedSparseGraph();
Vertex v1_copy = (Vertex) v1.copy(target);
Vertex v2_copy = (Vertex) v2.copy(target);
Edge e_copy = (Edge) e.copy(target);
```

The vertices `v1_copy` and `v2_copy` are considered to be "equal to" the vertices `v1` and `v2`, respectively, and the edge `e_copy` is equal to the edge `e`. As a convenience, the Java built-in `Object.equals` method has been defined in **JUNG** to respect this equivalence relation. Thus, for example, the statements

```
v1 == v1_copy.getEqualVertex(original);
v1.equals(v1_copy);
```

each evaluate to `true` in the context of the code given above.

There are some restrictions that govern when and where vertices and edges may be copied:

- A vertex or edge cannot be copied into a graph if it already has an equivalent in that graph. (This implies that the original graph and the target graph may not be the same.)
- The vertex/edge must pass any constraints that the target graph imposes on vertices/edges.

- The vertices incident to an edge must have equivalents in the target graph before the edge can be copied into that graph. (Thus, in the example above, we could not have copied the edge `e` until its incident vertices `v1` and `v2` had been copied.)

It can also be useful to be able to define a more general mapping between vertices (for example, if two different data sets include some of the same entities). For this purpose, **JUNG** provides the `VertexMapper` interface, which specifies a method that returns the vertex to which a specified vertex is mapped; it also includes several implementations of this interface, which use different mechanisms to establish the mapping (for example, string labels, vertex equality, and a hash table).

5. User Data

Users can associate data with graphs, edges, or vertices in two ways: by extending the **JUNG** classes, or by using the built-in **JUNG** user data repositories.

5.1. Class Extension

Users can extend the classes provided so that they include the variables/properties (and methods for manipulating those fields) that the user desires. This mechanism is most appropriate for applications which are designed to operate on a specific data set, each of whose elements have known properties. For instance, in the CiteSeer co-authorship network, each vertex might store the associated author's name, email address, and webpage (as shown in the example below).

The ability to extend the **JUNG** classes is a feature of Java, and is not specific to **JUNG**. The `AbstractSparse` classes use the built-in Java `Object.clone` method to copy instances of `Vertex`, `Edge`, and `Graph`; as a result, copies of such objects will duplicate only the object itself (the copied object thus has references to the same objects as the original).

This sample code creates a class for the representation of a CiteSeer author, by extending `SparseVertex` with additional author-specific information.

```
public class Author extends SparseVertex
{
    protected String name;
    protected String email_address;
    protected String webpage;

    public Author( String name, String email_address, String webpage )
    {
        this.name = name;
        this.email_address = email_address;
        this.website = webpage;
    }

    public String getName() { return name; }
    public String getEmailAddress() { return email_address; }
```

```
        public String getWebpage() { return webpage; }
    }
```

5.2. User Data Repositories

JUNG provides a built-in mechanism, the `UserData` class, for annotating graph elements with data. This mechanism is particularly useful for handling data which is either temporary or idiosyncratic (*i.e.*, data which not every graph element of that type will have or need). Examples of such data include a tag to indicate that a vertex has been selected in a visualization, a value that represents the output of an algorithm (which may be discarded as soon as it is no longer of interest), or a textual note that a data analyst attaches to a vertex of interest ("this author (P_Smith) may be the same as the author named P_Smyth").

Each **JUNG** object (graphs, vertices, and edges) has an associated user data repository; objects are stored in this repository with a specified retrieval key for quick access. Users may add, remove, modify, or fetch items from this repository; import data from another object's repository; list the objects in the repository; and check to see whether a given retrieval key is in the repository. Objects, when added, are given an associated "copy action" that determines how (or whether) the object is copied when imported into another repository.

This sample code shows how the user data repositories can be used to store and manipulate author-specific information.

```
String name_key = "name";
String email_key = "email";
String website_key = "website";
String selected = "selected";

// read in information from a file f
for (int i = 0; i < number_of_authors; i++)
{
    Vertex v = (Vertex) g.addVertex(new SparseVertex());
    v.addUserDatum(name_key, getNextName(f), UserData.SHARED);
    v.addUserDatum(email_key, getNextEmail(f), UserData.SHARED);
    v.addUserDatum(website_key, getNextWebsite(f), UserData.SHARED);
}

.
.
// vertex v has been selected in a visualization: mark it
v.addUserDatum(selected, selected, UserData.REMOVE);
// get name for v
String v_name = v.getUserDatum(name_key);
.
.
// v's email address has changed: update it
v.setUserDatum(email_key, "smyth@ics.uci.edu", UserData.SHARED);
// v has been deselected; remove this mark
```

```
    v.removeUserDatum(selected);
```

Although this example does not show it, the **JUNG** user data repository can contain any `Java` object, including other vertices, edges, or graphs.

When a graph element `a` is copied (with the `copy` method), the newly created element `b` attempts to copy each of the objects in `a`'s user data repository to `b`'s user data repository. The behavior of each such copy attempt will depend on the type of "copy action" that was specified when the corresponding user data element was created. **JUNG** provides three different copy action types: `UserData.CLONE`, `UserData.REMOVE`, and `UserData.SHARED`; each of these implements the method `onCopy`, which defines the behavior of a user data element when it is asked to provide a copy of itself.

`UserData.CLONE`'s version of `onCopy` returns a copy of the user datum, as defined by the `Java` `clone` method; `importUserData` then places this copy in the target graph element's user data repository. This clone is completely independent of the original. (If the user datum does not support the `clone` method, `onCopy` will throw the `Java` `CloneNotSupportedException`.)

`UserData.SHARED`'s version of `onCopy` returns a reference to the original user datum; `importUserData` then places this reference in the target graph element's user data repository. Thus, any changes to this user datum that are made by one of the graph elements that share this user datum will be reflected in all such graph elements.

`UserData.REMOVE`'s version of `onCopy` returns null; that is, user data that is created with this copy action will not be copied by the `copy` method.

5.3. Decorators, Indexers, and Labellers

JUNG includes a number of interfaces and classes (in the `graph.decorators` subpackage) that provide structured mechanisms for attaching data to **JUNG** elements. These are used in some of **JUNG**'s analytic algorithms (which may need to store or recover values, such as weights, for vertices or edges) and some of the visualization mechanisms; they can also be utilized by user code. A few examples follow.

The `Indexer` class contains methods that create a mapping between the vertices of a graph and the integers $\{0, 1, \ldots, n-1\}$ (where n is the number of vertices in the graph). It provides mechanisms to get the index of a given vertex (`getIndex(v)`) and to get the vertex with a specified index (`getVertex(i)`). Among other things, `Indexer` thus makes it convenient to arrange a set of vertices in an array, using each vertex's index as an index into the array.

`VertexStringer` is an interface that provides a method signature for retrieving the `String` label of a specified vertex. The `PluggableRenderer` class uses several different interfaces like this to access information on how each individual vertex and edge should be drawn.

The `StringLabeller` class implements `VertexStringer`, and is similar to `Indexer`; it provides facilities for fetching vertices given strings (labels) and vice versa. However, the labels are user-defined and thus need not follow any particular pattern. Vertices that have not been labelled simply will not be accessible by the indexer.

`NumberEdgeValue` is an interface that specifies method signatures for setting and getting a number associated with an edge. Since it is an interface rather than a class, the user may store this information in whatever fashion is most convenient (the **JUNG** user data repository, a special `Edge` implementation's fields, a user-created and -maintained mapping, etc.).

6. Filtering

The **JUNG** filtering mechanism removes selected vertices and/or edges from input graphs, and returns new graphs. For example, Figure 8 shows a subgraph `g` of the CiteSeer co-authorship network which was created by filtering out all vertices of distance <= 2 from a specified vertex; Figures 9 and 10 show the creation of a subgraph of `g`, where all vertices with degree less than 4 were removed.

These new graphs are copies of the original, containing all the same vertices and edges except for those that have been removed. A `Filter` takes in a `Graph` and returns an `UnassembledGraph`, which contains the individual vertices and edges that passed the `Filter`. Since there may be edges for which both endpoints did not pass the filter, an `UnassembledGraph` must be assembled into a new graph, which checks each edge to make sure that its endpoints are present before copying it into the new graph.

The assembly process can be slow, so it is sometimes desirable to string together several filters in a row, and not assemble the result until the last `Filter` has been run. This is done by creating a filter that implements the `EfficientFilter` interface. An `EfficientFilter` is a type of `Filter` that can filter an `UnassembledGraph`, and return another `UnassembledGraph`. A filter which examines structural properties of graphs is probably not appropriate to implement as an `EfficientFilter`, because `UnassembledGraphs` may contain incorrect topology information (in particular, as noted above, the edge set may include some ill-formed edges). It is the responsibility of the user to determine whether a given filtering mechanism can be implemented as an `EfficientFilter`.

While a user can write a custom filter merely by implementing the interface, it is often easiest to extend one of the two provided base `Filter` classes, `VertexAcceptFilter` and `EdgeAcceptFilter`. Each of these requires the user to implement a single method (`acceptVertex` or `acceptEdge`, respectively). By default, these are not declared to be `EfficientFilters`; however, users may certainly create extensions of these filters that are `EfficientFilters`.

The `SerialFilter` mechanism applies a series of filters sequentially to a specified graph, in the order in which they were added to the `SerialFilter`. As the filters are applied, it checks to see whether each one is an `EfficientFilter`, and calls `assemble` only when necessary.

The `LevelFilter` interface was designed to be used in conjunction with the `GraphDraw` mechanism (described in Section 9). `LevelFilters` are filters that take an integer parameter, which is used to determine the operation of the filter (for instance, filtering all edges with weight less than the value of this parameter). With a `LevelFilter`, a slider on a visualization can be tied directly into the `Filter`, and thus can allow the user to control this parameter directly, and generate a dynamically changing graph.

A user can create a `Filter` based on an existing `Predicate`, using the `EdgePredicateFilter` and `VertexPredicateFilter` classes provided; in this case, the edges/vertices that pass the filter are precisely those that pass the predicate. This can simplify the design of specialized filters.

7. Event Handling

Some applications generate extra processes (called "threads" in Java) so as to handle several different tasks concurrently; this generally includes any programs that provide a graphic user interface (GUI). If a graph can change over time, or in response to user input, this may result in synchronization problems (in which one thread is modifying the graph at the same time that another is trying to read it). To address this issue, **JUNG** provides a way for one thread to "listen" for changes to a graph (which may have been made by another thread).

The general pattern is that a object can register itself as a listener to certain types of events; this causes the object to be added to a list of such listeners, which are notified (via a call to a standardized method that the listener object must provide) when the event occurs.

Currently, **JUNG** can register classes (using the `ArchetypeGraph.addListener` method) to listen for graph mutation events: vertex addition and removal, and edge addition and removal. Mechanisms to allow classes to listen for changes to the user data repositories have been developed, and are planned for future release.

8. Algorithms

JUNG provides several different categories of network algorithms. A selection of them is listed here.

8.1. Ranking

Ranking algorithms assign values to each vertex (or edge) according to a set of criteria that reflect structural properties of the network. These criteria are generally intended to measure the "influence", "authority", or "centrality" of a given vertex/edge.

Several of these algorithms interpret their input as a Markov network: a directed weighted graph in which the vertices represent states, the edges represent possible state transitions, and the edge weights represent transition probabilities. (A directed graph with non-negative weights can be converted into a Markov network by normalizing the weights on the outgoing edges from each vertex to sum to 1.)

Ranking algorithms that **JUNG** provides include `BetweennessCentrality` (Brandes (2001)), which labels each vertex and edge in a graph with a value that is derived from the number of shortest paths that pass through it; `PageRank` (Page, Brin, Motwani, and Winograd (1998); Brin and Page (1998)), which ranks each vertex in a modified Markov network according to its stationary probability; `PageRankWithPriors` (White and Smyth (2003)), a generalization of `PageRank` whose ranks are calculated relative to a specified set of root vertices; `HITS` (Kleinberg (1999)), which ranks each vertex in a graph according to the "hubs-and-authorities" importance measures; `KStepMarkov` (White and Smyth (2003)), which ranks each vertex according to a fast approximation of the `PageRankWithPriors` algorithm; and `VoltageRanker` (Wu and Huberman (2004)), which treats the network as an electrical circuit (with specified voltage sources and sinks) and assigns each vertex a value corresponding to its residual voltage in this circuit. Figure 4 shows a visualization of a network whose vertices have been ranked by `VoltageRanker`.

8.2. Clustering and Blockmodelling

A cluster is a collection of objects that are all similar to each other in some way. In a network, similarity is often based on topological properties such as connectivity, but can also be based on the properties of vertices or edges in the network. Clustering algorithms provided by **JUNG**

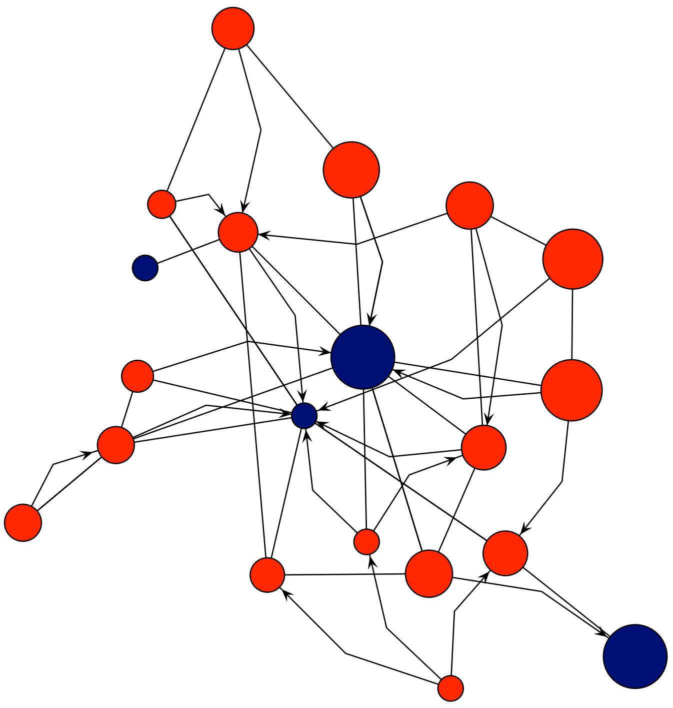


Figure 4: A random graph generated using the preferential attachment model (Barabasi and Albert (1999)). The vertices' sizes reflect the ranks generated by `VoltageRanker`; higher-ranked vertices are larger. Voltage source and sink vertices are colored dark blue, and all other vertices are colored red.

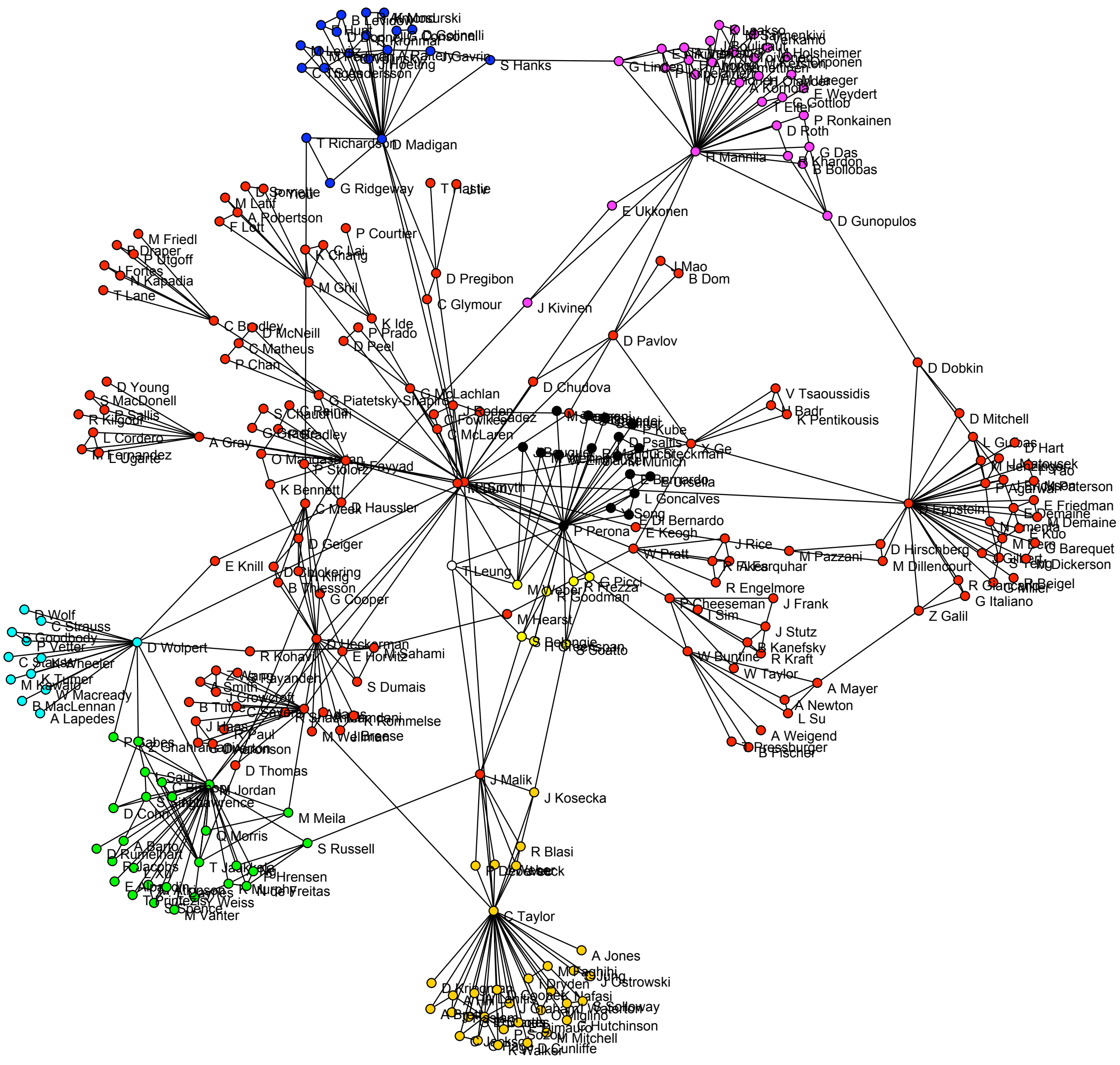


Figure 5: Clusters of a subgraph of the CiteSeer co-authorship graph, generated by `VoltageClusterer`. Each color represents a distinct cluster.

include `EdgeBetweennessClusterer` (Girvan and Newman (2002)), which computes clusters for a graph based on the betweenness property of the edges; `WeakComponentClusterer`, which finds all weak components in a given graph, where a weak component is defined as a (maximal) subgraph in which each pair of vertices is connected by at least one undirected path; and `VoltageClusterer` (Wu and Huberman (2004)), which assigns vertices to clusters based on their tendency to have similar voltages in the network, treated as an electrical circuit (see `VoltageRanker`, above). Figure 5 shows a subgraph of CiteSeer whose vertices have been clustered by `VoltageClusterer`, defined by the vertices within distance 2 of author "P_Smyth."

In the field of social network analysis, two vertices are said to be *structurally equivalent* if they are each adjacent to the same set of vertices. **JUNG** provides *blockmodelling* algorithms (Wasserman and Faust (1994)) which identify sets of structurally equivalent vertices. These algorithms can be useful for detecting possibly duplicate entities (for example, if a data set contains two or more copies of a single person, possibly with slightly differently spelled names), and for simplifying visualizations (structurally equivalent vertices may be represented visually as a single "supervertex", which can reduce the visual complexity and thus make the overall network easier to interpret). Figure 6 shows an example of such a simplified visualization.

8.3. Topology, Paths, and Flows

These algorithms perform operations on (and calculate properties of) graphs that relate to the graph's topology (that is, the structures and substructures formed by the ways that the vertices are linked together by edges). Topological algorithms that **JUNG** provides include `BFSDistanceLabeler`, which labels each vertex in a graph with the length of the shortest unweighted path from a specified vertex in that graph; `KNeighborhoodExtractor`, which returns the subgraph of a graph whose vertices are separated by no more than k edges from a specified vertex; `EdmondsKarpMaxFlow` (Edmonds and Karp (1972)), which labels each edge in a directed, edge-weighted graph with the flow along that edge which is consistent with the maximum flow for the graph; and `DijkstraShortestPath` (Dijkstra (1959)), which calculates the length of the shortest (weighted) paths from a specified vertex to other vertices in the same network.

8.4. Randomly Generated Networks and Statistics

There are a number of ways to characterize networks in terms of a specific topological property; some of these can be framed in terms of a generative probabilistic model, which can be used to guide the creation of a random graph. JUNG provides several such random graph generators, including `BarabasiAlbertGenerator` (Barabasi and Albert (1999)), which successively adds vertices to a graph using the preferential attachment model; `EppsteinPowerLawGenerator` (Eppstein and Wang (2002)), which creates a graph whose degree distribution follows the power law; and `KleinbergSmallWorldGenerator` (Kleinberg (2000)), which creates a graph which has the small-world property.

JUNG also includes several classes that calculate various statistical measures on graphs. These measures include degree distributions, clustering coefficients (the extent to which vertices' neighbors are connected to one another), and measures of discrete distribution similarity such as Kullback-Leibler divergence.

8.5. Transformations

It is sometimes necessary to convert a graph of one type to another; this can arise in a few different circumstances.

- Certain algorithms operate only on directed (or undirected) graphs. The `DirectionTransformer` class can transform any `Graph` into either an `DirectedGraph` or an `UndirectedGraph`; edges in the original graph are replaced with analogues of themselves that have the appropriate direction.

- One may wish to create a graph which is structurally distinct from the original, but preserves some of the information from the original; for example, one may wish to

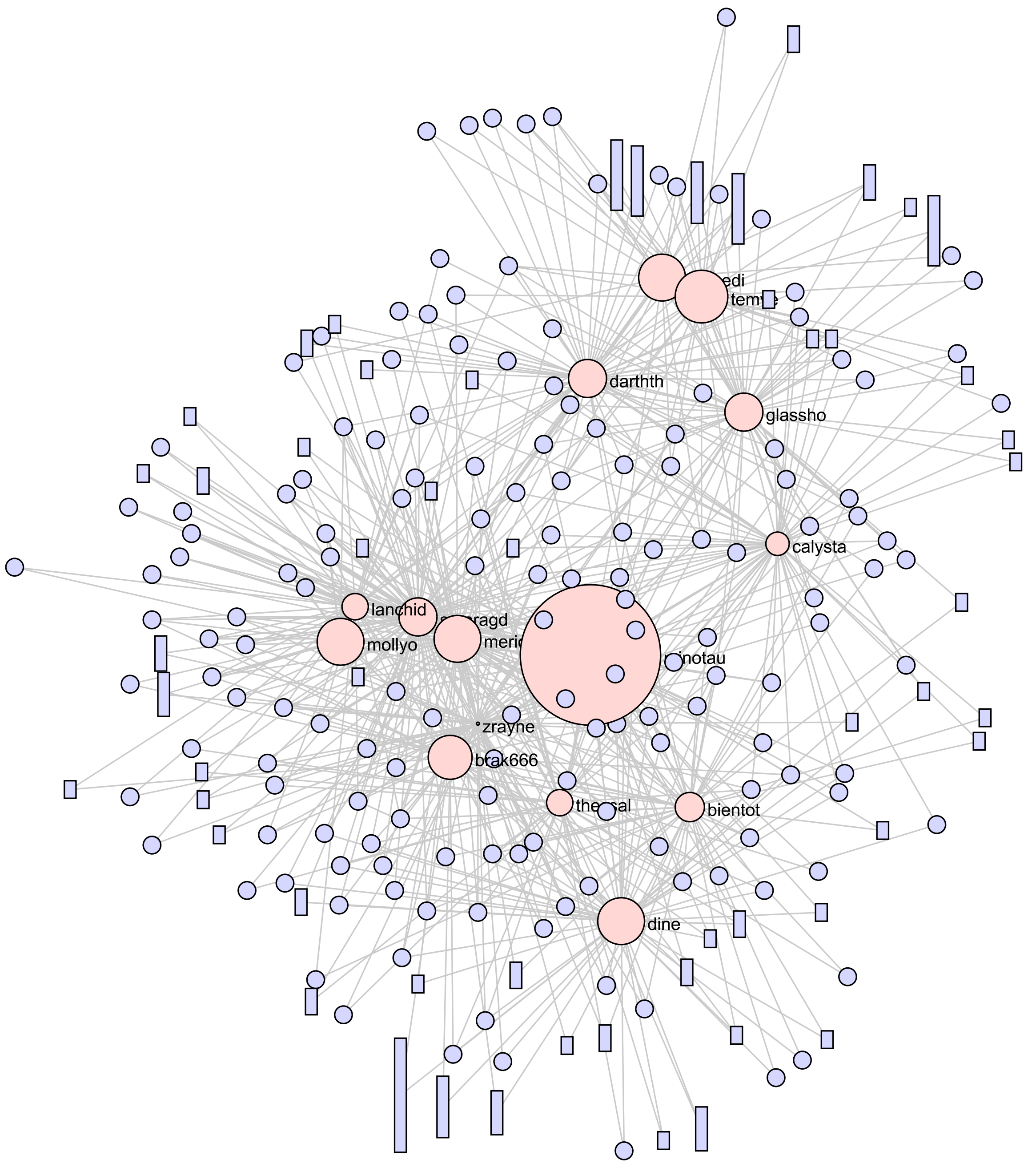


Figure 6: A network for which structurally equivalent vertices have been visually combined into vertical blocks, whose height represents the number of vertices in a block.

transform the CiteSeer two-mode researcher-paper bipartite graph (in which some vertices represent researchers, other vertices represent papers, and edges represent authorship) to a single-mode co-authorship graph (in which vertices represent researchers, and edges represent co-authorship relationships). The `KPartiteFolder` class can transform a `KPartiteGraph` into a unipartite `Graph` whose vertices are copies of those from a specified partition of the original graph, and whose edges represent "mediated" relationships in the original graph (that is, if the edges `(a,b)` and `(b,c)` exist in the original, then `(a,c)` exists in the transformed graph; multiple instances of such relationships can be represented in the transformed graph either by parallel edges or by decorations on a single edge).

- The process that resulted in the creation of a graph may not have identified all the details of the graph type; for example, a graph may be a k-partite graph in terms of its connectivity, but not have been created as an implementation of `KPartiteGraph`. The `KPartiteSparseGraph` class can construct a `KPartiteSparseGraph` that is a copy of an existing `Graph`, given an appropriate set of partition specifications that are known to apply to the graph. (That is, the original graph is not modified structurally; this construction only works if the original graph is actually k-partite.)

9. Visualization

JUNG provides mechanisms for laying out and rendering graphs. The current renderer implementations use the Java Swing API to display graphs, but they may be implemented using other toolkits (such as SWT).

In general, a visualization requires one of each of the following:

- A `Layout`, which takes a graph and determines the location at which each of its vertices will be drawn.
- A (Swing) `Component`, which provides a "drawing area" upon which the data is rendered. **JUNG** provides a `VisualizationViewer` class for this purpose, which is an extension of the Swing `JPanel` class. A currently available experimental version of `VisualizationViewer` allows the user to create a "window" on the graph visualization, which can be used to magnify (zoom in on) portions of the graph, and to select different areas for magnification (panning).
- A `Renderer`, which takes the data provided by the `Layout` and paints the vertices and edges into the provided Component.

JUNG provides the `GraphDraw` class, which provides a framework for the interaction of these components by packaging the `VisualizationViewer`, the `Renderer`, and the `Layout` together. The default implementation fetches the location of each vertex from the `Layout`, paints each one with the `Renderer` inside the Swing `Component`, and paints each edge as a straight line between its vertices. Users may customize this behavior as desired; **JUNG** includes utilities and support classes that facilitate such customization. For instance, `FadingVertexLayout` provides a mechanism that can be used to create fading effects when vertices are filtered out and subsequently restored; this can be useful for highlighting ongoing changes, such as may occur during the temporal evolution of a social network.

The `PluggableRenderer` class is an implementation of `Renderer` that provides a number of ways for the user to customize the way in which the graph is rendered, including the vertex shape, size, color, and label, and the edge color, thickness, and label; each of these properties can be specified for each individual vertex or edge. Some of these mechanisms are demonstrated in Figure 7.

Since **JUNG**'s data structures are completely separate from its visualization mechanisms, it is also possible to use other (`Java` or `Java`-compatible) visualization libraries to generate visualizations. One such library is **prefuse** (Heer, Card, and Landay (2005)); **JUNG** provides basic support for converting its data structures into their **prefuse** equivalents. (A `Java` applet on **JUNG**'s website demonstrates this capability.)

10. Input and Output

Graphs may be built programmatically using the techniques described in Section 4. However, in many cases, network analysts will have text files, or databases, which contain the data to be analyzed; in addition, analysts may wish to create a permanent record of a network that they have generated or modified. To these ends, JUNG supports several input and output formats, and provides general mechanisms that allow users to create their own formats.

10.1. Text Files

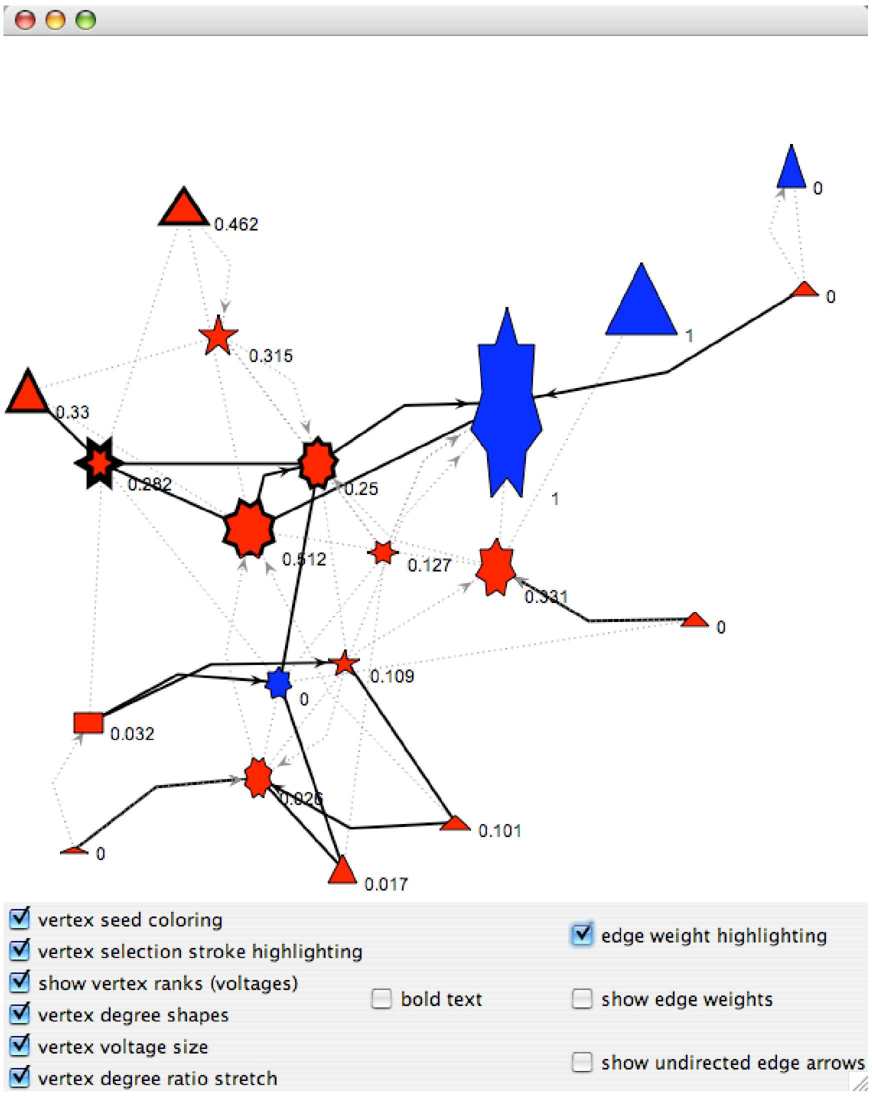
Pajek (Batagelj and Mrvar (2004)) is a tool that is popular in the social network analysis community, which has its own (complex) file format. **JUNG** can currently interpret the portions of the Pajek format that define directed, undirected, and mixed-type graphs; string (text) labels for vertices; and numeric edge weights. The format can also represent time-series and labeled vertex partition information, but **JUNG** does not as yet interpret these parts of the format.

GraphML (Brandes *et al.* (2004)) is an XML-based file format whose purpose is to provide a common file-based representation of network relationships. **JUNG** can currently interpret the portions of this format that define directed, undirected, and mixed-type graphs, and simple vertex and edge decorations. **JUNG** does not currently interpret the hypergraph and nested graph portions of the format.

Data is often provided in text files that are not in a common format. In such cases, developers have two choices: convert the text files into a common format, or write a parser for the existing format. Because there are many different possible text data formats, **JUNG** cannot provide a parser that works for all text data formats; however, the `jung.io` package does provide a few examples of parsers for non-canonical formats (`BipartiteGraphReader`, `MatrixFile`), which developers can use as models.

10.2. Databases

Network data may sometimes be stored in a database; this facilitates certain kinds of "flat" statistical analysis and retrieval, but is not conducive to most methods for network analysis. **JUNG** provides two different experimental demonstration mechanisms for manipulating



☑ vertex seed coloring
☑ vertex selection stroke highlighting
☑ show vertex ranks (voltages)
☑ vertex degree shapes
☑ vertex voltage size
☑ vertex degree ratio stretch
☐ bold text
☑ edge weight highlighting
☐ show edge weights
☐ show undirected edge arrows

Figure 7: A demonstration, using `PluggableRenderer`, of several different mechanisms for customizing a visualization.

network data which is stored in a database.

The first mechanism provides methods for reading the graph from the database into main memory, and for writing the graph (or portions of it) back to the database for long-term storage, if it has been changed; the program then manipulates this copy of the network, rather than addressing the database. This requires minimal effort on the part of the developer, but does mean that the graph has to fit in main memory; networks of several hundred thousand vertices have required approximately a gigabyte of memory (O'Madadhain *et al.* (2005)).

The second mechanism retains the network in the database; in this case, **JUNG** creates a collection of graph, vertex, and edge objects that refer directly to the database. This is the only practical method if the network is too large to fit in main memory, but operations are generally expected to be slower than those on a network which is loaded into memory. (In principle, caching and intelligent preloading could be used to reduce the fetching overhead, if only part of the network is being analyzed.) This mechanism requires database-specific implementations of the graph, vertex, and edge types, so that they know how to connect to the database to retrieve their metadata, as necessary.

11. Case Studies

In this section, we explore two scenarios in which **JUNG** has been used:

- the design of an agent-based simulation of a social network; this is based on a discussion with the developers of **Repast** (**Repast** Organization for Architecture and Development (ROAD) (2004)) on the best use of **JUNG** for this purpose.
- Netsight (Boey, O'Madadhain, and Smyth (2004)), a **JUNG**-based application for exploratory data analysis on network data sets.

11.1. Agent Simulations and Multimodal Graphs

Repast (Recursive Porous Agent Simulation) is a toolkit for creating agent-based models. In Fall 2004, the **Repast** developers asked the **JUNG** developers for guidance on how **JUNG** could be used to bolster **Repast**'s support for using social networks within such models. This discussion below is adapted from the dialog on **JUNG**'s web-based open discussion forum (O'Madadhain and Howe (2004)).

Repast agents may have one or more roles, and may participate in one or more types of interactions with other agents. Users of **Repast** may want to get information on just one of these modes, or on all modes. Since **Repast** is used to run agent simulations, the mechanisms which support such queries need to be fast. **JUNG** can represent such networks in several different ways; the best solution would depend on the size and complexity of the graph, and the operations to be used. Some possible representations include:

1. One single-modal graph, where modes are implicitly represented by decorations on vertices and edges (thus, an edge might have decorations on it that indicated that this edge represented a friendship, a supervisory relationship, and a collaborative relationship, in different contexts). This would be the most space-efficient, but would require extra time for mode-specific queries.

2. One multi-modal graph, where each vertex would represent a single mode of a given entity, and each edge a single mode of a relationship. This is less space-efficient, but can be more time-efficient for mode-specific queries. To minimize confusion, it might be useful to define equivalence classes that would identify the different vertices for a given entity, for example as all pertaining to that entity.

3. Multiple single-modal graphs. This is the least space-efficient–especially if there are multiple combinations of entities and relationships, each of which gets its own graph–but is probably the most time-efficient for mode-specific operations, although possibly less efficient than the first option for operations that operate on all modes.

JUNG can support each of these models; in addition, it can convert a single multi-modal graph into multiple single-modal graphs using filters. The built-in vertex and edge constraint mechanisms, and the predicate utilities, would also be useful for keeping track of the different modes. Another approach (currently under development) would be to write versions of `Vertex` and `Edge` that would filter their adjacency sets according to a specified predicate. This would allow users to do "on-the-fly" filtering without creating new graphs, thus saving space at the expense of time.

For a more specific example of such a network, consider a situation where the actors (agents) consist of banks, employees and customers, and in which actors' relationships are of the following types: patronage (customers patronize banks), employment, and friendship (employees may be friends with customers). As before, the best representation depends on the details, which in this case revolve around questions of uniqueness and separability.

If we assume that customers may patronize more than one bank, and that employees may work for more than one bank, and that no individuals are both customers and employees, then **JUNG**'s k-partite graph implementation (`KPartiteSparseGraph`) would be appropriate: the entity types would define the graph partitions, and each relationship would connect entities from two distinct partitions.

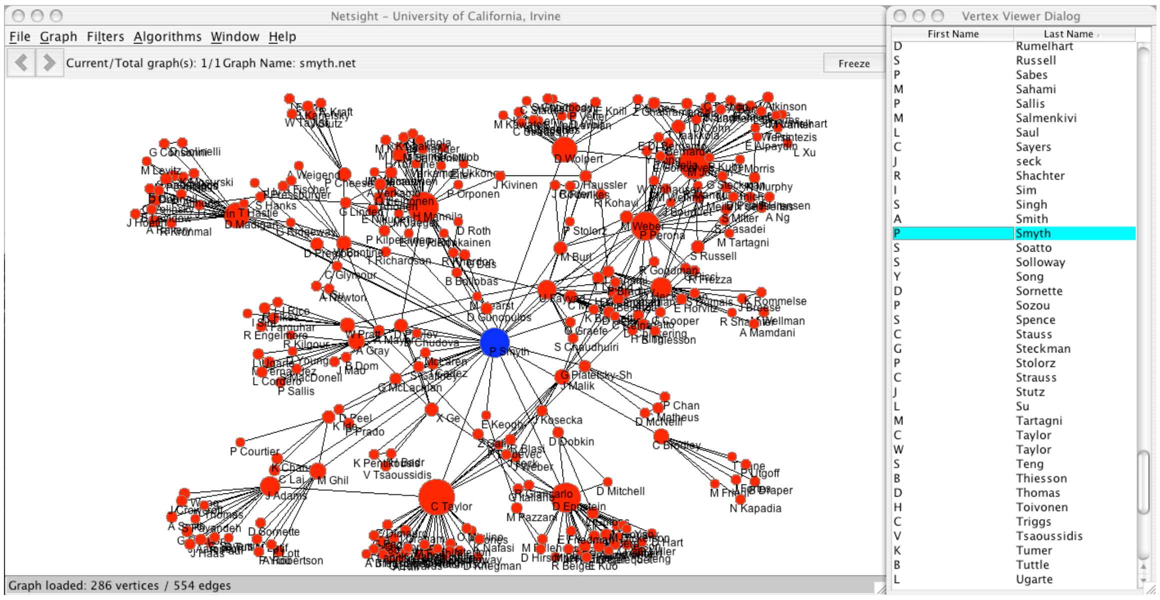
If, on the other hand, we assume that there is a one-to-one relationship between customers and banks, and between employees and banks, then the relationships could be represented implicitly by giving customers and employees a "bank" field, and attaching lists of customers and employees to each bank.

If employee/employee or customer/customer relationships are of interest, then the relationships no longer define a k-partite graph (since edges in a k-partite graph only connect entities of different partitions, *i.e.*, types). In this case, one would use a slight modification of the `KPartiteSparseGraph` class which had a different set of edge constraints.

Finally, if a single individual can be both a customer and an employee, then individuals may either be represented by multiple vertices (one vertex for each role, which may be problematic unless the roles are not expected to influence one another), or by a single vertex, with tags to indicate each of its roles.

11.2. Netsight

Netsight (Boey *et al.* (2004)) is a relatively simple visualization and analysis tool for large network data sets that has been developed at the University of California, Irvine, using the **JUNG** framework described in this paper.


Netsight – University of California, Irvine
File
Graph
Filters
Algorithms
Window
Help
Current/Total graph(s): 1/1 Graph Name: smyth.net
Freeze
Graph loaded: 286 vertices / 554 edges
Vertex Viewer Dialog

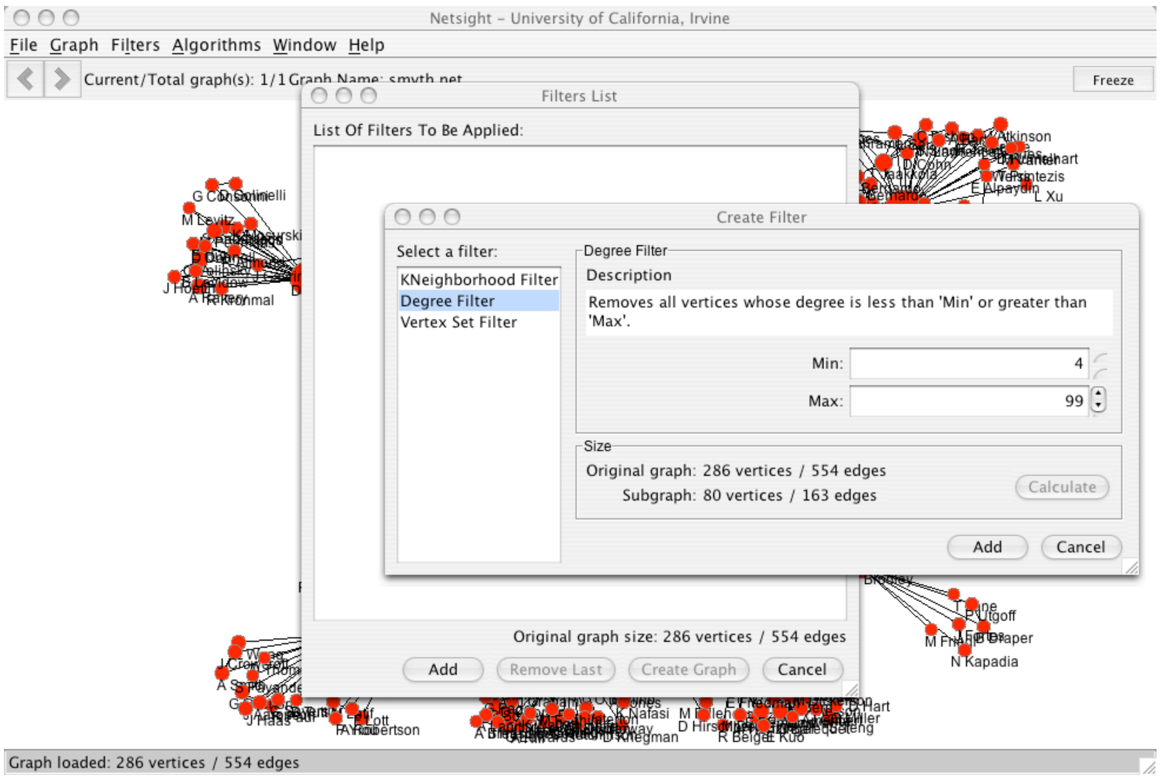
First Name	Last Name
D	Rumelhart
S	Russell
P	Sabes
M	Sahami
P	Sallis
M	Salmenkivi
L	Saul
C	Sayers
J	seck
R	Shachter
I	Sim
S	Singh
A	Smith
P	Smyth
S	Soatto
S	Solloway
Y	Song
D	Sornette
P	Sozou
S	Spence
C	Stauss
G	Steckman
P	Stolorz
C	Strauss
J	Stutz
L	Su
M	Tartagni
C	Taylor
W	Taylor
S	Teng
B	Thiesson
D	Thomas
H	Toivonen
C	Triggs
V	Tsaoussidis
K	Tumer
B	Tuttle
L	Ugarte

Figure 8: A portion of the CiteSeer co-authorship network, as visualized using Netsight. The currently selected vertex is colored blue, and highlighted in the vertex selector.

The tool was developed in the context of a project at the University of California, Irvine funded by the National Science Foundation which involved the development of algorithms for large relational data sets, such as CiteSeer (White and Smyth (2003), Steyvers, Smyth, Rosen-Zvi, and Griffiths (2004), Rosen-Zvi, Griffiths, Steyvers, and Smyth (2004)). In developing data analysis algorithms for such a data set it is quite useful to be able to visualize (e.g., via graph layout algorithms) relationships among authors. However, since the entire graph is so large, an effective method is needed for visualizing the local structure of different parts of this large graph. Thus, a tool was needed that could load the graph into main memory, allow a user to interactively select a subgraph of interest, and then produce a layout of that subgraph.

Standard network visualization tools such as **Pajek** and **UCINET** do not offer such an interactive filtering capability. Thus, the Netsight tool was developed specifically for this purpose. Netsight uses several different aspects of the **JUNG** libraries to facilitate exploratory data analysis on large networks. Some of its key features include:

- **decoupling of display and analysis**: Netsight provides the user a way to browse and select vertices (by label *e.g.*, name), filter the graph, and run analyses without visualizing the network; this can be crucial to the analysis of large networks.
- **filtering**: Netsight uses the **JUNG** filtering mechanisms to allow the network analyst to select a subgraph (or derived graph) to be analyzed and/or visualized. The available filters include degree filters (which remove vertices whose degrees do not lie in a specified range) and neighborhood filters (which remove vertices that are not within a specified distance from a specified vertex or set of vertices). The user may create composite filters which consist of a series of filters to be applied in sequence.
- **multiple graph navigation**: Netsight allows the analyst to load several networks, and


Netsight – University of California, Irvine
File Graph Filters Algorithms Window Help
Freeze
Filters List
Create Filter
Calculate
Add
Cancel
Add
Remove Last
Create Graph
Cancel

Current/Total graph(s): 1/1 Graph Name: smyth.net

List Of Filters To Be Applied:

Select a filter:
KNeighborhood Filter
Degree Filter
Vertex Set Filter

Degree Filter
Description
Removes all vertices whose degree is less than 'Min' or greater than 'Max'.

Min: 4
Max: 99

Size
Original graph: 286 vertices / 554 edges
Subgraph: 80 vertices / 163 edges

Original graph size: 286 vertices / 554 edges

Graph loaded: 286 vertices / 554 edges

Figure 9: Netsight's filtering interface.

to create subgraphs of any loaded or derived network (subject to the restriction imposed by the amount of available memory). The analyst navigates among these networks using a tree-based interface (where a network is listed as a "child" of the network from which it was derived, if any).

- **algorithms**: Netsight provides interfaces to several of **JUNG**'s algorithms for social network analysis, including ranking algorithms such as PageRank. It has also been used to provide a visual interface to the output of a link formation model (O'Madadhain *et al.* (2005)).

12. Related Work

JUNG was created out of a perceived need for a general, flexible, and powerful API for manipulating, analyzing, and visualizing graphs and networks. There exist numerous other packages and tools for visualizing and manipulating networks; we present here a comparison of **JUNG** with the following: **UCINET** (Borgatti, Everett, and Freeman (2004)), **Pajek** (Batagelj and Mrvar (2004)), R (R Development Core Team (2004)) with **sna** (Butts (2004)), and **GFC** (IBM Corporation (1999)). Other similar software (which we will not discuss in detail for reasons of space) includes NetMiner (Cyram Company, Ltd. (2004)), StOCNET (Boer, Huisman, Snijders, and Zeggelink (2003)), MultiNet (Richards and Seary (2004)), InfoVis (Fekete (2004)), InfoVis Cyberinfrastructure (Penumarthy, Mane, and Börner (2004)), Visone

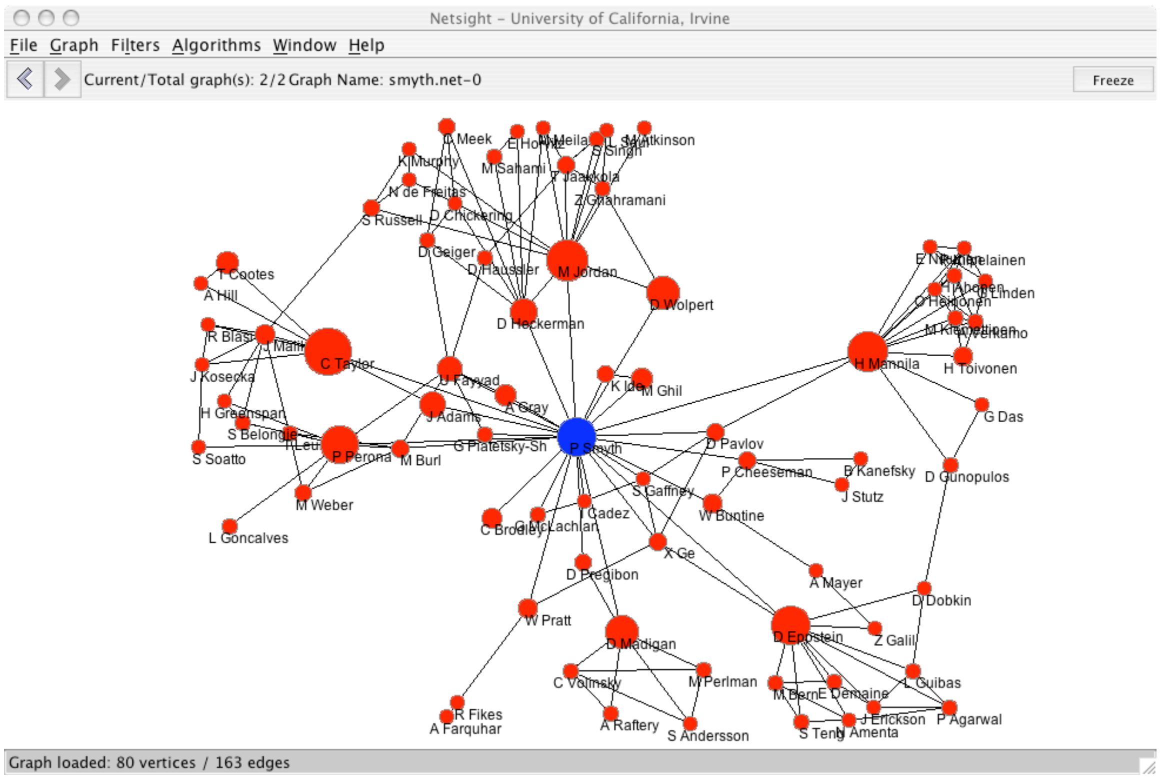

Netsight – University of California, Irvine
File Graph Filters Algorithms Window Help
Current/Total graph(s): 2/2 Graph Name: smyth.net-0
Freeze
Graph loaded: 80 vertices / 163 edges

Figure 10: The result of applying the filter from Figure 9 to the graph in Figure 8.

(Brandes and Wagner (2003)), Boost (Siek, Lee, and Lumsdaine (2001)), JGraph (Alder (2005)), and yFiles (yWorks (2004)).

UCINET and **Pajek** are stand-alone applications that each provide a number of tools for visualizing and analyzing networks. However, they cannot be conveniently addressed programmatically by other applications, so they are not well-suited to process large numbers of graphs. Furthermore, they are applications rather than libraries, so users cannot write their own routines; this is particularly problematic for complex data sets (whose complexities may not have been anticipated by the application's designers) and for analysts that wish to use methods that are not provided as part of the application. Finally, they are closed-source projects, and executables are only available for the Windows environment.

R is a specialized programming language designed primarily for statistical computing and graphics. The **sna** package extends R in somewhat the same way that **JUNG** extends Java; like **JUNG**, **sna** provides a number of routines for social network analysis and visualization (including some 3-D visualization). R is well-suited for the rapid development of scripts and (in conjunction with **sna**) for on-the-fly network analysis, especially for analysis which requires sophisticated statistical tools. In addition, while it is not technically platform-independent, R is available on several popular platforms (so code written in R is reasonably portable), and is open-source. However, R and **sna** do have some limitations, particularly for large sparse networks:

- R passes copies of a routine's arguments to it; this requires considerably more space and time than passing references to these arguments (as Java does), especially for large objects.

- R does not have a native sparse matrix format (although a sparse matrix package (Koenker and Ng (2003)) does exist, **sna** does not make use of it), which can greatly increase the space and time required.

These limitations can be addressed by implementing some of the critical portions of the code in C/C++, but at the cost of additional complexity of code and reduced portability. As a result, R and **sna** are generally not the most efficient way of manipulating or analyzing networks of more than a few thousand entities. By contrast, while Java is not as well-suited to rapid script development as R, it does provide a convenient platform for the development of more complex tools (for example, those with graphic user interfaces, database connectivity, and/or Web support), and since **JUNG**'s representations and algorithms are both space- and time-efficient, Java and **JUNG** are more natively scaleable.

GFC is a Java graph drawing-oriented API. It can only use Java's AWT/Swing mechanisms for rendering, contains few algorithms for network analysis, is no longer actively supported, and is not open-source. (In this, it is similar to a number of other network APIs.)

13. Future Directions

JUNG currently provides many of the tools and elements that are most commonly required for writing software that manipulates, analyzes, and visualizes network data sets. Future releases are planned to include the following features, several of which are currently under development. These features should significantly expand the set of available tools and enhance users' abilities to write robust code.

- Expansion of the input and output options, including full implementation of database connectivity (as discussed in Section 10.2), ability to parse more complex **Pajek** and GraphML files, and support for other network file formats, such as the .dl and DOT formats defined by the **UCINET** and **Graphviz** (Ellson and North (2004)) projects.

- Providing additional analysis tools: algorithms, statistical tests, parameter estimations, etc.

- Support for on-the-fly filtering (mentioned in Section 11.1); this could make working with subgraphs much less memory-intensive.

- New visualization architecture: the current visualization system flexibly supports a broad set of visualizations, but is limited in certain respects (such as its ability to accommodate dynamic and evolving graphs, or to save and restore visual states). Continuing work on **JUNG** is investigating ways to adapt or replace the current system.

- Creating a standard XML-based representation of **JUNG**'s algorithms which specify their parameter and return types, as well as brief textual descriptions. This would allow writers of applications such as Netsight to expand the algorithms which it provides by editing a configuration file, rather than modifying the code itself.

14. Acknowledgements

We would like to thank the Open Source Technology Group (2005) for their hosting of this project on SourceForge, and the Eclipse Foundation (2005) for providing the Java Eclipse IDE; these free services and tools allowed us to concentrate on development rather than infrastructure.

This material is based upon work that was supported in part by the National Science Foundation under the Knowledge Discovery and Dissemination (KD-D) Program under Grant No. IIS-0083489 (authors JO, PS, and YB), by the NSF under Grant Nos. IIS-0133749, IIS-0205724 and IIS-0326105 (author DF), by a GAANN fellowship (author JO), and by an NSDEG fellowship (author SW).

References

Adar E, Feinberg D (2005). “GUESS: Graph Exploration System.” `http://www.hpl.hp.com/research/idl/projects/graphs/`.

Alder G (2005). “JGraph: The Java Graph Visualization Library.” `http://www.jgraph.com`.

Apache Jakarta Project (2004). “Commons Collections.” `http://jakarta.apache.org/commons/collections/`.

Apache XML Project (2004). “Xerces2 Java Parser.” `http://xml.apache.org/xerces2-j/index.html`.

Barabasi AL, Albert R (1999). “Emergence of scaling in random networks.” *Science*, **286**, 509–512.

Batagelj V, Mrvar A (2004). “Pajek: Program for Large Network Analysis.” `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`.

Battista GD, Eades P, Tamassia R, Tollis IG (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

Boer P, Huisman M, Snijders TAB, Zeggelink EP (2003). “StOCNET: an open software system for the advanced statistical analysis of social networks.” `http://stat.gamma.rug.nl/stocnet`. Version 1.4. Gronigen: ProGAMMA / ICS.

Boey YB, O’Madadhain J, Smyth P (2004). “Netsight.” `http://jung.sourceforge.net/netsight`.

Borgatti S, Everett M, Freeman L (2004). “UCINET: Software for Social Network Analysis.” `http://www.analytictech.com/ucinet`.

Brandes U (2001). “A Faster Algorithm for Betweenness Centrality.” *Journal of Mathematical Sociology*, **25**(2), 163–177.

Brandes U, Eiglsperger M, Kaufmann M, Lerner J (2004). “The GraphML File Format.” `http://graphml.graphdrawing.org/`.

Brandes U, Wagner D (2003). "visone - Analysis and Visualization of Social Networks." In M Jünger, P Mutzel (eds.), "Graph Drawing Software," pp. 321–340. Springer-Verlag.

Brin S, Page L (1998). "The anatomy of a large-scale hypertextual Web search engine." *Computer Networks and ISDN Systems*, **30**(1–7), 107–117. URL `citeseer.ist.psu.edu/brin98anatomy.html`.

Butts C (2003). "Network Inference, Error, and Informant (In)Accuracy: A Bayesian Approach." *Social Networks*, **25**(2), 103–140.

Butts C (2004). "S Routines for Social Network Analysis in the R Environment: The SNA Package." `http://erzuli.ss.uci.edu/R.stuff/`.

CERN (2004). "The Colt Project." `http://dsd.lbl.gov/~hoschek/colt/`.

Cyram Company, Ltd (2004). "NetMiner." `http://www.netminer.com`. Version 2.5.

Dijkstra EW (1959). "A note on two problems in connexion with graphs." *Numerische Mathematik*, **1**, 269–271.

Eclipse Foundation (2001). "SWT: The Standard Widget Toolkit." `http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`.

Eclipse Foundation (2005). "The Eclipse Integrated Development Environment." `http://eclipse.org/`.

Edmonds J, Karp R (1972). "Theoretical improvements in algorithmic efficiency for network flow problems." *Journal of the ACM*, **19**(2), 248–264.

Ellson J, North S (2004). "Graphviz." `http://www.graphviz.org`.

Eppstein D, Wang JY (2002). "A steady state model for graph power laws." In "2nd International Workshop on Web Dynamics," URL `citeseer.ist.psu.edu/eppstein02steady.html`.

Fekete JD (2004). "The InfoViz Toolkit." In "Proceedings of the 10th IEEE Symposium on Information Visualization (InfoVis'04)," pp. 167–174. IEEE Press.

Fisher D, Dourish P (2004). "Social and temporal structures in everyday collaboration." In "CHI '04: Proceedings of the 2004 Conference on Human Factors in Computing Systems," pp. 551–558. ACM Press. ISBN 1-58113-702-8.

Froehlich J, Dourish P (2004). "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams." In "Proceedings of the International Conference on Software Engineering," pp. 387–396.

Girvan M, Newman M (2002). "Community structure in social and biological networks." *Proceedings of the National Academy of Sciences*, **99**(12), 7821–7826. URL `http://www.pnas.org/cgi/content/abstract/99/12/7821`.

Heer J, Card SK, Landay JA (2005). "**prefuse**: a toolkit for interactive information visualization." In "CHI '05: Proceedings of the 2005 Conference on Human Factors in Computing Systems," .

Hoff P, Raftery A, Handcock M (2002). "Latent space approaches to social network analysis." *Journal of the American Statistical Association*, **97**, 1090–1098.

Huisman M, Snijders TA (2003). "Statistical analysis of longitudinal network data with changing composition." *Sociological Methods & Research*, **32**, 253–287.

IBM Corporation (1999). "GFC: Graph Foundation Classes for Java." http://www.alphaworks.ibm.com/tech/gfc.

Kleinberg JM (1999). "Authoritative sources in a hyperlinked environment." *Journal of the ACM*, **46**(5), 604–632. URL http://citeseer.ist.psu.edu/kleinberg99authoritative.html.

Kleinberg JM (2000). "The small-world phenomenon: An algorithmic perspective." In "Proceedings of the 32nd ACM Symposium on Theory of Computing," .

Koenker R, Ng P (2003). "**SparseM**: Sparse Linear Algebra for R." http://www.econ.uiuc.edu/~roger/research/sparse/sparse.html.

Lang DT (2004). "The R And S Interface to **Omegahat** and Java." http://www.omegahat.org/RSJava/.

Lawrence S, Giles CL, Bollacker K (1999). "Digital Libraries and Autonomous Citation Indexing." *Computer*, **32**(6), 67–71. ISSN 0018-9162.

Newman M (2000). "Who is the best connected scientist? A study of scientific coauthorship networks." http://www.santafe.edu/sfi/publications/wpabstract/200012064.

O'Madadhain J, Howe T (2004). https://sourceforge.net/forum/forum.php?thread_id=1164328\&forum_id=252062.

O'Madadhain J, Smyth P, Adamic L (2005). "Learning Predictive Models for Link Formation." To be presented at the International Sunbelt Social Network Conference.

Open Source Technology Group (2005). "SourceForge.net." http://sourceforge.net.

Page L, Brin S, Motwani R, Winograd T (1998). "The PageRank Citation Ranking: Bringing Order to the Web." *Technical report*, Stanford Digital Library Technologies Project. URL citeseer.ist.psu.edu/page98pagerank.html.

Penumarthy S, Mane KK, Börner K (2004). "InfoVis Cyberinfrastructure Software Framework." http://iv.slis.indiana.edu/sw/.

R Development Core Team (2004). *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria. 3-900051-07-0, URL http://www.R-project.org.

Repast Organization for Architecture and Development (ROAD) (2004). "**Repast**: Recursive Porous Agent Simulation Toolkit." http://repast.sourceforge.net.

Richards W, Seary A (2004). http://www.sfu.ca/~richards/Multinet/Pages/multinet.htm. Version 4.55.

Rosen-Zvi M, Griffiths T, Steyvers M, Smyth P (2004). "The Author-Topic Model for Authors and Documents." In M Chickering, J Halpern (eds.), "Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence," pp. 487–494. Banff,Canada.

Siek JG, Lee LQ, Lumsdaine A (2001). *The Boost Graph Library User Guide and Reference Manual*. C++ In-Depth Series. Addison-Wesley.

Smith MA, Fisher D (2005). "Social Network Visualization of Usenet Newsgroups." To be presented at the International Sunbelt Social Network Conference.

Steyvers M, Smyth P, Rosen-Zvi M, Griffiths T (2004). "Probabilistic Author-Topic Models for Information Discovery." In "Proceedings of the Tenth ACM International Conference on Knowledge Discovery and Data Mining," pp. 306–315. ACM Press, Seattle, WA.

Sun Microsystems (2003). "Java Native Interface." http://java.sun.com/j2se/1.3/docs/guide/jni/.

Sun Microsystems (2004). "Creating a GUI with JFC/Swing." http://java.sun.com/docs/books/tutorial/uiswing/.

Wasserman S, Faust K (1994). *Social Network Analysis*. Cambridge University Press, Cambridge.

Watts DJ, Strogatz SH (1998). "Collective dynamics of 'small-world' networks." *Nature*, **393**, 440–442.

White S, Smyth P (2003). "Algorithms for estimating relative importance in networks." In "Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining," pp. 266–275. ACM Press.

Wu F, Huberman B (2004). "Discovering Communities in Linear Time: A Physics Approach." *Eur. Phys. Journal*, **B38**, 331–338.

yWorks (2004). "yFiles." http://www.yworks.com/en/products_yfiles_about.htm.

Affiliation:

Joshua O'Madadhain
Department of Computer Science
University of California, Irvine
Irvine, CA, USA
E-mail: jmadden@ics.uci.edu
URL: http://www.ics.uci.edu/~jmadden

Journal of Statistical Software *Submitted:* yyyy-mm-dd
MMMMMM YYYY, Volume VV, Issue II. *Accepted:* yyyy-mm-dd
http://www.jstatsoft.org/