Working Paper Series
ISSN 1177-777X

Graph-RAT Programming Environment

Daniel McEnnis

Working Paper: 2009/04
August 12, 2009

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Graph-RAT Programming Environment

Daniel McEnnis
University of Waikato, Hamilton, New Zealand
dm75@waikato.ac.nz

August 12, 2009

Abstract

Graph-RAT is a new programming environment specializing in relational data mining. It incorporates a number of different techniques into a single framework for data collection, data cleaning, propositionalization, and analysis. The language is functional where algorithms are executed over arbitrary sub-graphs of the data. Analytical results can be conducted using collaborative filtering or machine learning techniques. The example algorithms are under BSD license.

1 Introduction

Graph-RAT is a language and framework that draws from the same traditions as Weka, Matlab, and GATE for providing a high-level interface for performing a wide range of sophisticated analysis without requiring knowledge of the underlying interpretation. However, Graph-RAT differs in that it sees only relational data. All data has a context—such as the table that it is stored in and its relationships with other tables. All data is placed in a context and all operations are relative to a context within a graph paradigm. This makes programming in Graph-RAT considerably different to any other programming environment.

In the traditions of GATE, Graph-RAT provides an end-to-end computing environment. It includes data collection, data cleaning, data manipulation, propositionalization, and (by Weka) machine learning primitives. A complete set of algorithms can be found in the technical report [6] which extends the algorithm list published the year before [5].

Contrary to other languages, Graph-RAT makes the assumptions of dependency between algorithms and data. All programs in Graph-RAT are explicitly tied to the specific data structures of the data they utilizes. However, each algorithm component is as general as possible. This creates a functional language where components are typically simple operations and programs compositions involving these components.

The structure of Graph-RAT is quite different from other languages and is discussed in detail in the section 2. The data collection and importing capabilities are discussed in section 3. Section 4 discusses the similarities and

differences between Graph-RAT and other languages. Section 5.2 discusses how to construct applications in Graph-RAT using the command-line interpreter. Section **??** discusses how to construct and embed Graph-RAT code in a Java program.

2 Structure

Graph-RAT is unique in the way that it unifies a number of different analysis techniques and methods into a single high-level analysis language. The basic structure has been published before in [5] and [6]. It was also demoed at ISMIR [4]. The technical report in particular contains a complete summary of the algorithms available, so it will not be repeated here.

The graph consists of three distinct levels of abstraction. The lowest level is the data structure. The next higher level of abstraction is the graph-view tree. The final level of abstraction is the algorithm pipe.

2.1 Data Structure

The building blocks of graphs are properties. Each property is a (possibly multi-valued) variable that contains a Java object. Every property exists within a context. This context is defined by vertices (actors) and edges (links). Each vertices has a class (mode) as does each edge (a relation). Graphs themselves can also have typed properties. Any Java class is an acceptable property type, provided the appropriate serialization routines are provided. A collection of actors and links are collected within a graph, indexing them.

Actors are created using an ActorFactory singleton which uses a java.util.Properties object to specify the parameters to create the Actor. See the Javadoc of ActorFactory for a list of all actor types, their properties, and what these properties influence. Likewise, Links are created from a LinkFactory, and Graphs from a GraphFactory.

2.2 View Tree

The set of all actors and links form the root view of a graph. All graphs can have an arbitrary number of child graphs which contain a strict subset of the root graph's actors and links. This forms an acyclic hierarchy of subsets of the original data in the form of a B-Tree. This tree can be queried using a regular expression for which the set of closest-to-root graphs whose ID matches the expression is returned.

Subgraphs are either created manually by adding existing actors and links to a new graph object or by use of the getSubGraph() method of the Graph interface. Once actors and links are added, the graph can be listed as a child of a graph by registering it with the addChild() method of the Graph interface. Similarly, getSubGraph creates a graph which can be added with the addChild() method.

2.3 Language Structure

A Graph-RAT program consists of three components: A graph declaration, a sequence of data acquisition modules, and a sequence of algorithm—view expression pairs.

The graph declaration that determines which graph implementation will be used during execution. The parameters are fed to a factory which generates the base graph of the given type. For SQL-backed graphs, this may load a pre-existing graph without data acquisition modules.

The next component is a sequence of data acquisition algorithms that add properties to the graph. Each data acquisition algorithm entry must have a unique ID and is executed sequentially using the parameters provided.

The third component is a sequence of view expression—algorithm pairs. For each pair, the expression is used to obtain a sequence of graphs. The given algorithm is then executed with each graph sequentially using the declared parameters. Missing parameters are filled with the default value.

2.4 Algorithm Types

Graph-RAT algorithms can be divided into three primary types—propositional algorithms, clustering algorithms, and analytical algorithms.

The most basic building block algorithms are the propositional algorithms. These algorithms transform properties into weka Instance objects and move these properties into different parts of the graph using aggregator functions. Ultimately, one can propositionalize the entire data set using only these methods.

Clustering algorithms take a given set and alter the graph view tree based on the content of the data. The only properties created during this process are those that declare the graph ID of the new sub-tree an actor or link has been assigned to.

Analytical algorithms take a set of graph data and construct new actors or links which are added back to the graph. These algorithms include link-based analysis such as prestige analysis and actor based analysis such as machine-learning algorithms.

2.5 Data Source Abstraction

All algorithms and data acquisition modules execute against the Graph interface. Because of this, their can be a number of different implementations of this interface. Currently, there are 3 major implementations: MemGraph (stored in memory), Derby SQL, and Postgresql SQL. There are also a number of minor graph types for specific circumstances such as the actor graph (collects the ID's of all actors) and the null graph (never stores anything).

3 DataCollectionCapabilities

Graph-RAT is designed to make easier the collection and loading of relational data. There are three different ways to load relational data. One is to construct a database in the required table structure and register this database with Graph-RAT. Another is to create an XML file with the correct data. The final is to use a built-in multi-threaded web-crawling system to acquire the data from the internet dynamically.

The structure of an SQL database description of Graph-RAT involves the following tables: Graph, Actor, Link, GraphActor, GraphLink, GraphProperty, ActorProperty, and LinkProperty. These tables can be created automatically setting the Initialize parameter true when creating a graph using the GraphFactory.

The MemGraph file can be a quite compact way of describing relational data. By default, the reader assumes that the file is gzipped. Each type of data that can be stored in a graph is defined within this format. See http://graph-rat.sf.net for full documentation of the syntax and examples.

The final method involves use of the crawler interface. By default, the interface makes no assumptions about the protocol for accessing data: crawling for data on a file system uses the same syntax as crawling for data on the Internet.

The built-in web-crawler uses an extensible system for parsing web-pages. When creating the crawler, a set of parsers are registered. These parsers take the document and a reference to a graph and perform arbitrary analysis over the document. Inside the parser, there may be references to "spider" new documents. This differs from traditional crawlers as parsers can discriminate between important links and unimportant (i.e. navigation) links. Furthermore, it is possible to have only a subset of parsers used to analyze a single document when a crawl request is made to the crawler. Screen scraping web pages and the relationships between them is made much easier through these features.

The following code demonstrates setting up a crawl of the LastFM web services:

```
Crawler crawler = new WebCrawler(); LastFMUserExpansion expansion
= new LastFMUserExpansion(); crawler.setParsers(expansion.seteUpParsers(<file
of usernames>,<graph object>));
```

currently, due to a bug in the system, each of the first six parsers must manually setParser the ToFileParser (parser index 7 obtained from the crawler).

4 Language Comparisons

4.1 Graph-RAT vs GATE

There are a number of similarities between these two systems. Both provide a library of algorithms for building custom applications. However, GATE [2] uses a very different data structure than Graph-RAT. The GATE structure consists of a set of tags applied to a document where a tag has a start and a end. These

tags are equivalent to properties in Graph-RAT, but are treated as an ordered sequence (by start character). Like Graph-RAT, later algorithms consume tags from an earlier algorithm.

4.2 Graph-RAT vs Matlab

Similar to Matlab [3], Graph-RAT provides a library of algorithms to execute against its only data type - the graph. Graph-RAT has sets of properties that are roughly analogous to Matlab's matrix variables. Also, the algorithms themselves are designed to be extremely high-level, which is also typical of Matlab algorithms. Unlike Matlab, however, all variables are explicitly tied to a single data structure—every variable has an explicit context through which it can be related to other variables.

4.3 Graph-RAT vs Scheme

While Scheme [7] uses a binary tree to store all variables in an application, Graph-RAT uses a more general cyclic-graph as its most basic unit. Like Scheme, Graph-RAT has a tree of views roughly analogous to function calls in Scheme. The propositionalizing algorithms in Graph-RAT function as Scheme's cons and car functions. Likewise, the programming style of Graph-RAT is also primarily functional where the output of one algorithm is fed to others without any side-effects or other modifications of the graph.

4.4 Graph-RAT vs SQL

All of the operations of the graph interface can be re-written in terms of SQL statements. Furthermore, the propositional algorithms in Graph-RAT provide a limited query capability that correspond to a primitive form of SELECT statement in SQL. However, the propositional algorithms perform avoids 'not applicable' nulls and other problems involved in a traditional SELECT statement involving joins. Contrary to SQL, however, these SELECT statements modify the underlying tables - explicitly placing the results of the algorithms back into the database. Finally, there are no pure reads in Graph-RAT—every read implies a corresponding insert statement.

4.5 Graph-RAT vs LINQ

LINQ [1] is a programming environment in C# and is the most similar environment to Graph-RAT. While Graph-RAT focuses on model manipulation and set-based descriptions of the underlying data, LINQ provides a query-based view of the underlying data, focusing on analyzing combinations of sequences rather than combinations of sets. However, in LINQ, queries involving multiple queries still fall back on SQL-style joins to create sequences. Graph-RAT provides no equivalent query method, but provides a number of additional tools for creating subsets of the original data and analyzing them.

4.6 Graph-RAT vs Weka

Like Weka [8], Graph-RAT provides data acquisition from databases, input from XML, and data cleaning capabilities. Also, Graph-RAT provides a pipeline for performing experiments in a similar way to Weka. However, while Weka is designed to work on non-relational data, Graph-RAT is explicitly designed to work on relational data. As a result, many of the base components of Graph-RAT have no correlary in Weka. In fact, most of Weka—classification and clustering in particular—are included in Graph-RAT as individual algorithms operating on Instance objects on a set of actors.

4.7 Graph-RAT vs Other Graph Libraries

Table 1 presents a brief summary of prominent toolkits. The Hypergraph, Piccolo, Giny , Guess, Jung2, Prefuse, and JGraph toolkits are all geared towards visualization of graphs without any analysis capabilities. Prefuse contains a matrix-based descriptor that is particularly inefficient with different data-types over different modes, but does support arbitrary properties for an actor. Jung2 provides a set of analysis tools, but does not have built-in support for actors, modes, or properties, though its Template-based structure allows these features to be added later. The biggest drawback is that its general structure does not allow for propositionalization of relational data. Proximity provides support for modes and relations, but does not include an algorithm base and has limited support for paths and properties. Jena provides a sophisticated query language over multiple modes and relations but does not provide analysis capabilities in its language. Its focus on RDF, instead of a more compact XML, can also be a drawback.

Table 1: Existing graph toolkits

Name	URL	Description
Hypergraph	`hypergraph.sourceforge.net`	Hyperbolic tree applet for visualizing graph st in an XML file
Piccolo	`www.cs.umd.edu/hcil/piccolo`	Toolkit for 2D graphics, especially zooomable
prefuse	`www.prefuse.org`	Toolkit for data visualization including graphs
Giny	`csbi.sourceforge.net`	framework for interactive information visualiza
Guess	`www.graphexploration.cond.org`	analysis and visualization tool for graphs; uses prefuse, and Jung. Includes Jython interpreter
Jung2	`jung.sourceforge.net`	framework for modeling, analyzing and visuali graph-based data
JGraph	`www.jgraph.com/jgraph.html`	graph visualization component
Proximity	kdl.cs.umass.edu/software	graph analysis package
Jena	`jena.sourceforge.net`	RDF analysis language

5 Programming in Graph-RAT

While abstract descriptions of Graph-RAT are useful, demonstrating how to construct applications in Graph-RAT is also quite useful. The construction consists of three parts: choosing a graph, selecting data acquisition modules, and selecting algorithm modules.

To start with, one should use a simple graph. If the application's only function is web crawling, a UserGraph or other speciality graph is appropriate. Otherwise, choose the graph which best suites the type of data. If the content is limited in size, collecting in a MemGraph and saving to file may be the most appropriate. Otherwise, using one of the SQL-backed graphs would be a good choice as they are quite scalable.

5.1 Choosing a Data Acquisition Module

Deciding on a data acquisition module determines what analysis can be performed later. In other languages, this part is an offline, hand-done process. In Graph-RAT, data acquisition

5.1.1 Using ReadLastFMProfile

Unfortunately, acquiring data using the crawler is now disabled due to an upgrade of the LastFM web services (upgrade of Graph-RAT in progress). However, it is still possible to read the results of a previous run. (Contact the author for a copy under a creative commons non-commercial share and share alike license for a copy of the data.) This creates a graph consisting of Artist, User, and Tag actor modes. Artists have Tag relations with Tags, Users have Tag relations with tags, Users have UserArtistTags with Artists, Users have Friends with other Users, and Users have ListensTo relations with Artists. Artists have jAudio-derived content vectors as properties. Users have profile information as properties. This model is used in Section 5.2 as a demonstration of how to program in Graph-RAT.

5.1.2 Using CrawlLiveJournal

LiveJournal creates, by default, one mode—Users. They have properties from the FOAF itself and also have Friends relations with each other.

5.1.3 Using MemGraphReader

This reads an arbitrary MemGraph XML file and loads the XML file into memory. The modes, relations, and properties are entirely dependent on the content of the file.

Figure 1: Beginning example data model

5.1.4 Writing a New Data Acquisition Module

Any relational data can be mapped directly into the Graph-RAT data model. When importing an SQL table, the table types are actor modes. The foreign keys represent relations between actors. The tuples in a table represent properties. Each entry in the table represents a single actor instance whose links are defined by whatever foreign keys are in that entry.

The built-in crawler operates in a breadth-first search. With appropriate parsers, performing targeted web crawls are easy and efficient. When performing crawls, one of mode of actors are typically the web site with its contents described as properties. The links in the page are usually relations to other pages. Frequently, terms are separated out and treated as actors as well.

5.2 Algorithm Examples

All of the following examples of Graph-RAT algorithm pairs are from the domain of music recommendation created by the CrawlLastFM data acquisition module. Figure 1 shows the modes and relations in this system.

The properties are as follows:

Actor Node	Property Name	Property Type
User	(Profile Items)	Various Strings and Integers
Artist	SongVector	double array

In all of these examples, the relation 'ListensTo' describes the ground truth. The purpose of each algorithm is to create a new relation 'Derived' which attempts to recreate the 'ListensTo' relation as completely as possible. For this purpose, all algorithms start with the following code:

```
   <Scheduler>
<Graph>
<Name>MemGraph</Name>
</Graph>
<DataAquisition>
<Name>ReadLastFMProfile</Name>
</DataAquisition>
<DataAquisition>
<Name>ReadAudioFiles</Name>
<Properties/>
</DataAquisition>
```

and ends with:

```
    <Algorithm>
<Pattern>.*</Pattern>
<Class>Kendall Tau</Class>
```

```
<Name>Evaluation</Name>  <Property>
<Name>GroundTruth</Name>
<Value>ListensTo</Value>
</Property>
</Properties>
</Algorithm>
</Scheduler>
```

5.2.1 Local Recommendation

The first application is quite simple—the equivalent of a 'hello world' GraphRAT application. It recommends what each friend listens to. It does this by aggregating over ListensTo relations, aggregating across Friends relation, then the LinkToProperty to created the Derived Link.

```
   <Algorithm>
<Name>AggregateByLink</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Friend</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>data</Value>
</Property>
<Property>
<Name>aggregator</Name>
<Value>nz....reusablecores.aggregator.Concatenate</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>PropertyToLink</Name>
<Property>
<Name>source</Name>
<Value>data</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Derived</Value>
</Property>
</Algorithm>
```

5.2.2 Collaborative Filtering

User-to-user collaborative filtering and Item-to-Item collaborative filtering algorithms re-create the majority of the collaborative filtering algorithms commonly used in both general recommendation algorithms as well as most social-based commercial recommenders.

The user-to-user algorithm calculates similarity between users then tries to recreate ListensTo by the same method as local recommendation.

```
    <Algorithm>
<Name>SimilarityByLink</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>ListensTo</Value>
</Property>
<Property>
<Name>destinationRelation</Name>
<Value>Similarity</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateByLink</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Friend</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>tagVector</Value>
</Property>
<Property>
<Name>aggregator</Name>
<Value>nz....aggregator.Concatenate</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>PropertyToLink</Name>
<Property>
<Name>source</Name>
```

<Value>data</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Derived</Value>
</Property>
</Algorithm>

Item-to-item calculates similarity between artists, then tries to predict ListensTo for a particular artist by aggregating the total correlations for each other artist a user listens to.

<Algorithm>
<Name>SimilarityByLink</Name>
<Property>
<Name>actorType</Name>
<Value>Artist</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>ListensTo</Value>
</Property>
<Property>
<Name>destinationRelation</Name>
<Value>Similarity</Value>
</Property>
</Algorithm> <Algorithm>
<Name>AggregateByLink</Name>
<Property>
<Name>actorType</Name>
<Value>Artist</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Similarity</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>relationships</Value>
</Property>
<Property>
<Name>aggregator</Name>
<Value>nz....aggregator.Concatenate</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateByLink</Name>
<Property>

```
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>ListensTo</Value>
</Property>
<Property>
<Name>source</Name>
<Value>data</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>data</Value>
</Property>
<Property>
<Name>aggregator</Name>
<Value>nz....reusablecores.aggregator.Concatenate</Value>
</Property>
</Algorithm>
<Name>PropertyToLink</Name>
<Property>
<Name>source</Name>
<Value>data</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Derived</Value>
</Property>
</Algorithm>
```

5.2.3 Direct Machine Learning

This algorithm propositionalizes all available data using aggregators, then uses traditional machine learning algorithms to predict the ListensTo groundtruth.

```
    <Algorithm>
<Name>ReadAudioFiles</Name>
<Property>
<Name>musicMode</Name>
<Value>Artist</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateByLinkProperty</Name>
<Property>
```

```
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>listensTo</Value>
</Property>
<Property>
<Name>innerAggregator</Name>
<Value>nz...Average</Value>
</Property>
<Property>
<Name>outerAggregator</Name>
<Value>nz...Average</Value>
</Property>
<Property>
<Name>source</Name>
<Value>songVector</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>songData</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateByLink</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>Friend</Value>
</Property>
<Property>
<Name>aggregator</Name>
<Value>nz...Concatenate</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>friendData</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateOnActor</Name>
<Property>
```

```
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>innerAggregator</Name>
<Value>nz...FirstItem</Value>
</Property>
<Property>
<Name>outerAggregator</Name>
<Value>nz...Concatenate</Value>
</Property>
<Property>
<Name>actorProperty</Name>
<Value>data</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>WekaClassifierMultiAttribute</Name>
<Property>
<Name>wekaClass</Name>
<Value>Adaboost</Value>
</Property>
<Property>
<Name>linkAbsence</Name>
<Value>false</Value>
</Property>
<Property>
<Name>actorProperty</Name>
<Value>data</Value>
</Property>
</Algorithm>
```

Each of these algorithms can be subdivided so the predictive strength of each component of the source data can be determined individually.

5.2.4 Machine Learning with Cultural Data

This algorithm clusters users according to their friends. Every user is then provided a vector describing the link structure of the cluster that it is in—providing cultural data for analysis.

Inserted at 5.2.3

```
 <Algorithm>
<Name>WekaClassifierClusterer</Name>
<Property>
<Name>actorProperty</Name>
<Value>friendData</Value>
```

```
</Property>
</Algorithm>
<Algorithm>
<Name>DirectedTriples</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>GraphToActor</Name>
<Property>
<Name>innerAggregator</Name>
<Value>nz...First</Value>
</Property>
<Property>
<Name>outerAggregator</Name>
<Value>nz...First</Value>
</Property>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>actorProperty</Name>
<Value>Triple</Value>
</Property>
</Algorithm>
```

5.2.5 Machine Learning with Orthogonalization

The previous algorithms assume that all artists are unrelated to one another—something that is blatantly not true. This algorithm corrects for this using PCA to create a vector for each artist which can be used instead of a straight link.

```
    <Algorithm>
<Name>SimilarityByLink</Name>
<Property>
<Name>actorType</Name>
<Value>Artist</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>ListensTo</Value>
</Property>
<Property>
```

```
<Name>destinationRelation</Name>
<Value>Similarity</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>DistanceMatrix</Name>
<Property>
<Name>actorType</Name>
<Value>Artist</Value>
</Property>
<Property>
<Name>actorVectors</Name>
<Value>true</Value>
</Property>
<Property>
<Name>actorProperty</Name>
<Value>artistVector</Value>
</Property>
</Algorithm>
<Algorithm>
<Name>AggregateByLinkProperty</Name>
<Property>
<Name>actorType</Name>
<Value>User</Value>
</Property>
<Property>
<Name>relation</Name>
<Value>listensTo</Value>
</Property>
<Property>
<Name>innerAggregator</Name>
<Value>nz...Average</Value>
</Property>
<Property>
<Name>outerAggregator</Name>
<Value>nz...Average</Value>
</Property>
<Property>
<Name>source</Name>
<Value>artistVector</Value>
</Property>
<Property>
<Name>destination</Name>
<Value>artistData</Value>
</Property>
</Algorithm>
```

```
<Algorithm>
<Name>WekaClassifierMultiAttribute</Name>
<Property>
<Name>wekaClass</Name>
<Value>Adaboost</Value>
</Property>
<Property>
<Name>linkAbsence</Name>
<Value>false</Value>
</Property>
<Property>
<Name>actorProperty</Name>
<Value>artistData</Value>
</Property>
</Algorithm>
```

These algorithms only scratch the surface but do give an idea of how to write applications in Graph-RAT.

5.3 Algorithms

The algorithms available are a super-set of the algorithms described in [6]. Beyond these are two new aggregators and principal component analysis.

6 Embed

These operations can also be embedded in Java. The scheduler, algorithms, and graph objects all have factories for creating them. The 'Name' XML property is entered as the 'Name' property on a java.util.Properties object. See each entries javadoc on `graph-rat.sourceforge.net` for a description of each property available.

7 Conclusions

Graph-RAT provides a feature-rich programming language for performing relational data mining. Every aspect of the process from aquisition to data cleaning to analysis to evaluation is presented within a single framework. A wide variety of algorithms can be created and evaluated quickly, making it an good prototyping environment for quite complicated analysis.

8 Future Work

Additional algorithms for separating testing and training relations are under way. Furthermore, a number of toolkit upgrades involving dynamic loading of

components, replacing Properties objects with Graph-RAT Property objects, and a Graph implementation that also conforms to the Actor interface.

9 Acknowledgments

The author would like to thank the University of Waikato for its generous support through the Waikato Doctoral Scholarship.

References

[1] J. Albahari and B. Albahari. *LINQ pocket reference*. O'Reilly Media, Sebastopol, CA, 2008.

[2] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of ACL*, 2002.

[3] MathwWorks. Matlab - the lanugage of technical computing, December 9. http://www.mathworks.com/products/matlab.

[4] D. McEnnis. Graph-rat: A mir toolbox. *International Conference of Music Information Retreival*, September 2008.

[5] D. McEnnis. Towards a music recommendation infrastructure. *New Zealand Computer Science Student Research Conference*, April 2009.

[6] D. McEnnis and D. Bainbridge. Graph-rat: Combining data sources in music recommendation systems. Technical Report 07-2008, Universitry of Waikato, Hamilton, New Zealand, July 2008.

[7] G. Springer and D. P. Freidman. *Scheme and the Art of Programming*. MIT Press, 1989.

[8] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.