Discovering Entity Correlations between Data Schema via Structural Analysis

by

Ashish Mishra

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified By .... [illegible]
Dr. Amar Gupta
Co-Director, Productivity From Information Technology (PROFIT) Initiative
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Discovering Entity Correlations between Data Schema via Structural Analysis

by

Ashish Mishra

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

Abstract

At the forefront of data interoperability is the issue of semantic translation; that is, interpretation of the elements, attributes, and values contained in data. Systems which do not adhere to pre-defined semantics in their data representations need to dynamically mediate communication between each other, and an essential part of this mediation is structural analysis of data representations in the respective data domains. When mediating XML data between domains, one cannot rely solely on semantic similarities of tags and/or the data content of elements to establish associations between related elements. To complement these associations one can build on relationships based on the respective domain structures, and the position and relationships of evaluated elements within these structures. A structural analysis algorithm uses associations discovered by other analysis, discovering further links which could not have been found by purely static examination of the elements and their aggregate content. A number of methodologies are presented by which the algorithm maximizes the number of relevant mappings or associations derived from XML structures. The paper concludes with comparative results obtained using these methodologies.

Thesis Supervisor: Dr. Amar Gupta
Title: Co-Director, Productivity From Information Technology (PROFIT) Initiative

Acknowledgments

My supervisor at MITRE, Michael Ripley, was a huge contributor throughout this project — he was a voice for suggestions and criticism, an ear to bounce ideas off, and a push at the right times when the going got slow. Additionally, my thesis supervisor, Dr. Amar Gupta, gave both direction and support in all phases of my research. Much of this thesis draws from a paper the three of us co-authored in 2002.

The foundation for the X-Map project was laid by David Wang, whose thesis was also a source for much of my work. My colleague, Eddie Byon, was a major contributor to the coding process, as well as helpful in discussing ideas and more mundane matters such as transportation.

Other helpers and motivators included Doug Norman at MITRE; Brian Purville, Danny Lai, and Andrew Chen at MIT; and Anne Hunter, the wonderful department administrator. Most of all, I owe this to my parents, without whom none of this would have been remotely possible.

Contents

Chapter 1

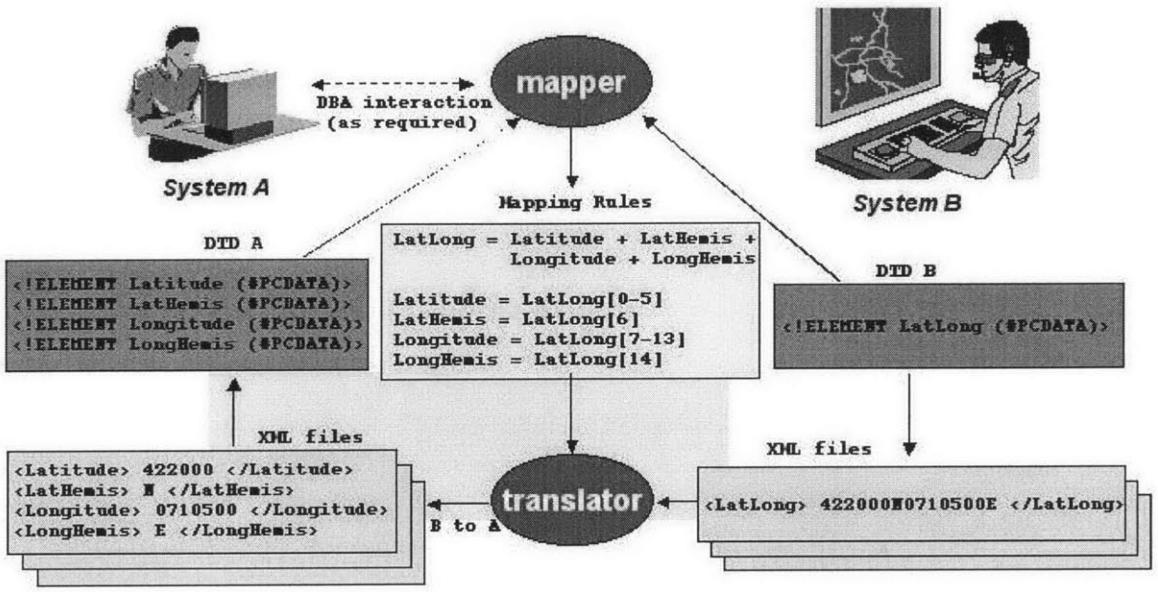
Context for Structural Analysis

For effective transfer of information between diverse processes in a networked environment, it is necessary that there be a known and agreed-upon protocol for interconversion between the "languages" they speak. In other words, the data structures they incorporate must agree on how they map between each other. Explicitly requiring the systems to agree on a data format for exchange is not always desirable as it forces constraints on their data structures and induces interdependencies between the systems involved. Enabling interoperability via a static mediator has various drawbacks, as either the mediator or the systems will constantly need to be updated as one of the processes' internal structure changes or new data domains are added. An alternate approach involves more sophisticated mediators that dynamically or statically adapt translations between systems, without interfering with their internal data formats.

A data communication language assumed for the domains is the eXtensible Markup Language (XML) [33]. XML is a flexible meta-language useful as a framework for interoperability: it immediately resolves several issues, such as parsing and character encoding recognition. Documents conforming to valid XML immediately reveal the logical structure of the data they contain, freeing a designer to focus on the analysis itself. XML also incorporates several technologies that can be leveraged in mediating interoperability: these include the XML Linking Language (XLink) [42], which aids in drawing arbitrary associations between resources; XML Schema, a schema description language for XML documents [58, 59, 60]; and XML Stylesheet Language Transforms (XSLT), a language for specifying transformations between XML documents [61]. These tools are very useful in describing and carrying out the process of conversion or translation. What is significantly lacking [34], however, is a process of actually discovering associations between documents and semi-automatically generating the transform scripts which can rearrange and modify the associated data. Requiring constant human intervention at this stage nullifies much of the advantage of having a dynamic, sophisticated mediator.

1.1 X-Map

Our research [5] has concluded that in order to effectively mediate between heterogeneous domains in XML, the following architecture is needed. Given two non-identical domains, two types of applications are required to meet these goals. The first application, the mapper, interrogates the data structures of the source and destination and creates the mappings between them. This application requires some user interaction, since all semantic differences in data cannot be automatically discovered. The second application, the mediator, operates on instance data to create a tailored product using the mapping rules. The beauty of this design is that once the mapping rules are found, the mediator applications can use the same set of mappings to process the instance data over again, as long as the structures are not changed. And, when the structures do change, the mapper can update the mappings before the mediator begins processing, thus providing current mappings to the mediator for the new instance data. Lastly, since the mediator simply reads in the mapping files, no code changes are necessary to operate on the new data structures. This architecture is depicted in Figure 1-1 [5].



```
<!ELEMENT Latitude (#PCDATA)>
<!ELEMENT LatHemis (#PCDATA)>
<!ELEMENT Longitude (#PCDATA)>
<!ELEMENT LongHemis (#PCDATA)>
```

```
LatLong = Latitude + LatHemis +
          Longitude + LongHemis

Latitude = LatLong[0-5]
LatHemis = LatLong[6]
Longitude = LatLong[7-13]
LongHemis = LatLong[14]
```

```
<!ELEMENT LatLong (#PCDATA)>
```

```
<Latitude> 422000 </Latitude>
<LatHemis> N </LatHemis>
<Longitude> 0710500 </Longitude>
<LongHemis> E </LongHemis>
```

```
<LatLong> 422000N0710500E </LatLong>
```

Figure 1-1: Mapping and Transformation Architecture

The X-Map system for semi-automated semantic correlation seeks to mediate XML data interoperability between multiple heterogenous systems. X-Map would play a significant role in the Joint Battle InfoSphere (JBI), which aims to provide information superiority to US Armed Forces. The JBI adheres to a publish and subscribe mechanism, which facilitates the sharing of information between interested parties, manipulation of information to create knowledge, and dissemination of that new knowledge to other interested parties upon creation. the interested parties send information via messages asynchronously back and forth. The role that fuselets play in the JBI amounts to enhancing and automating all three functionalities of the JBI; Information Input, Information Manipulation, and Knowledge Output and Interaction. That is, fuselets fuse information from several subscribed sources into knowledge.

X-Map's essential design was developed as part of the 'Fuselet Architecture for Adaptive Information Structures' project at MITRE corporation. The goal of the X-Map project is to enable adaptive data interoperability between participating systems through a transparent mediator. Among the key information components required for this task are (a) knowledge of possible mappings between elements in data structures, (b) a semi-automated means to derive these mappings, as well as to represent and store them so they can be reused by other applications, (c) transform functions to facilitate the mediation of data between systems, given knowledge of mappings between them, as well as to represent the transformations done by the mediation so they can be easily reused. X-Map's role is in the the first two tasks; semi-automatically discovering mappings and relationships, as well as storing information about them.

Key information components required by X-Map are:

1. Knowledge of possible mappings between elements in data structures,
2. The means to represent mappings and store them so they can be reused by other applications, and
3. Transformation functions that perform the specific operations required to transform an XML document from one domain to another.

As part of our research [5], the following components were developed:

1. An XLink based association language for representing the mappings,
2. An XSLT based language for representing the transformation functions, and
3. The specification of several different types of mediators that perform various mediation tasks.

This work has been described in [34, 35] and will not be further detailed here.

In X-Map, data about two XML domains are read in, analyzed, and the resulting mappings and transformation functions are generated as output. This design segmented the process of discovering mappings into two separate parts: equivalence analysis, which examines the tags, attributes, and data for similarities in form and content, and structural analysis, which looks at the structures of the XML files for similarities. After analyzing for potential mappings, X-Map displays to the user a graphical representation of the nominations it has made. As part of the User Interface (UI), the user can interrogate the system for information about the nominations, and can then accept or reject any of them. Afterwards, the UI allows the user to add mappings that may have been missed by X-Map. The user also needs to confirm the type of mapping nominated so the proper transformation functions can be created. After all user input is entered, X-Map outputs an X-Map Linkbase (the name given to a linkbase that contains mappings as opposed to hypertext linkings) of the mappings that exist and a set of transformation functions that encode how the mappings are performed. A diagram of the design developed for X-Map is shown in Figure 1-2 [5].

1.2 The Association Engine

The technical backbone of X-Map is the automated engine through which associations are discovered and nominated. This is the mechanism via which associations are discovered and captured by the association language. The general case of automatic association is very hard and implies the solving of consistent semantic matching of elements between two data-domains. The goal here is less ambitious: it is the use of semi-automated methods to make consistent semantic matches in suitably constrained cases, and to provide an approach that can make these matches within a specified margin of error with minimal re-

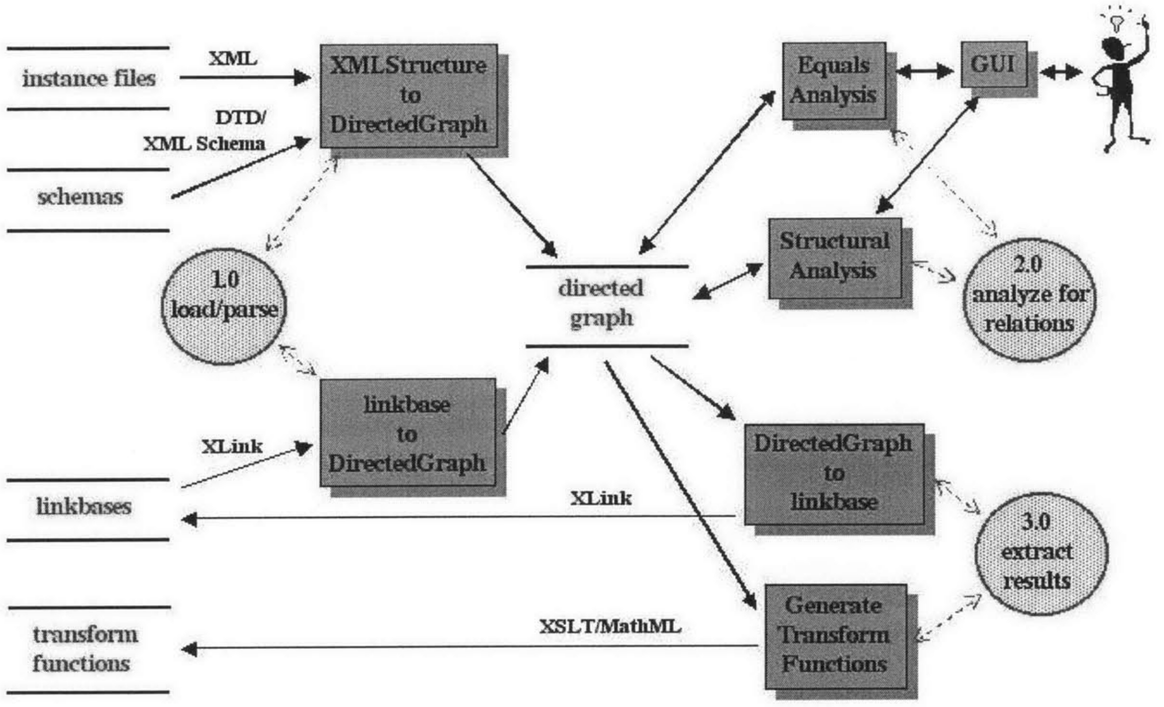


Figure 1-2: X-Map design

quirement of human intervention. The system will try its best to come up with appropriate semantic matches, and when it cannot provide a match or has serious doubts, or for ultimate validation, it will ask a human operator for assistance. Obviously we would like to minimize the work that has to be performed by the human operator; this implies nominating as many genuine mappings as possible while keeping to a minimum the incidence of "false positives" which require an operator to invalidate nominated associations.

Table 1.1 [5] illustrates some of the common types of associations that may occur, and examples of these. Although some of these use the "equivalence" relationship (defined in [34] as expressing essentially the same physical object or concept, and differing only in representation) as a fundamental building block, others do not. Note that not all of these relationships need be two-way: sometimes a one-way relationship implies an entirely different relationship (or no relationship) in the opposite direction.

Relation Type	XML Example		Description
Equals	`<EmpNo>`	`<EmpNo>`	both the tag names and data are the same
Synonym	`<SSN>`	`<SocialSecNo>`	the tag names are not the same, but the data is
Conversion	`<Weight units="kg">`	`<Weight units="lbs">`	data domain is the same, but a conversion is required
LookUp	`<CountryCode>`	`<CountryName>`	data domain is the same, but some type of code is used
Reordering	`<Date format="ddmmyy">`	`<Date format="mmddyy">`	data domain is the same, but reordering is required
IsLike	`<Title>`	`<JobDescription>`	data from one domain is "good enough" for the other
Generalization/Specialization			
— SuperType	`<TechnicalPaper>`		data from one domain is a superset of the other
— SubType		`<ConferencePaper>`	data from one domain is a subset of data from the other
Abstraction	`<DailySales>`	`<MonthlySales>`	data from one domain combined into a single value in the other
Aggregation/Decomposition			
— PartOf	`<FName>` `<MName>` `<LName>`		data from one domain is concatenated in the other domain
— Composite		`<Name>`	data from one domain takes multiple fields the other
Defaults	No element exists, always enter "1"	`<Quantity>`	default values needed in the other domain

Table 1.1: Generic Mappings

In many cases, the nature of the relationship(s) can be read off simply by observing the names and attributes of elements in the disparate domains. The most obvious instance arises when precisely the same element occurs in both domains. Alternately, it may be possible to establish a relationship by scanning for element names in a lookup table: for instance, the X-Map prototype supports using a lexicon of abbreviations, jargon or even inter-language translations. In cases where both of the above approaches fail, the algorithm tries simple string manipulations to compare element names: checking whether one is an acronym for the other, or simply differently spelled. As the scope of possible nominations increases, one eventually arrives at a point of trade-off between false positives and misses. The notion of a "score" describing a degree of confidence in the nominated association, as illustrated later in this paper, provides an elegant way of handling this trade-off.

The next step up involves examining instances of data in elements being compared. As before, the algorithm uses lookup tables, or scans the data instances for some of the relationships described earlier: precision differences, scaling factors (indicating different units being used), acronyms and abbreviations. There will again exist a trade-off between false positives and misses, depending on the scope of the analysis.

The above two techniques are termed "Equivalence Analysis" and "Data Analysis" respectively. This kind of static analysis is invaluable, especially initially when no other possibility is open. However there will invariably exist relationships which cannot possibly be picked up by the earlier techniques. An instance of this is the "`Employee-SalesPerson`" association described earlier in Table 1.1. To a human, it is clear that these are related: a `SalesPerson` is an example of an `Employee`. However, there is no semantic relationship between these element names for an automated agent to pick up. It is likely that the data immediately stored under this tag (as opposed to under child tags) are either unrelated, or the relationship does not get picked up by data analysis for whatever reason (e.g. sufficiently different formats). This would be a case where analysis of the data structures comes into play in nominating associations.

It follows that structural analysis and comparison of data domains is a key component in semi-automated detection of semantic mappings. The remainder of this paper describes implementations of this analysis, as tested in the X-Map prototype. The analysis depends heavily on our internal representation of the structure of these domains, so to begin with, this representation is described in greater detail. Subsequent sections describe different modes of resolution of this problem. The fundamental question under study may be approached from several different directions, as follows:

1. As a straightforward problem of developing heuristics for speculating on associations;
2. As a graph theoretic labeling problem: discovering isomorphisms between graphs representing the data structures; or
3. As a search space/optimization problem: minimizing the deviance between speculated and genuine mappings.

These different ways of looking at the problem shape different approaches toward solving it. Instances of several different types of techniques were implemented to compare and contrast their respective performances on sample data.

Chapter 2

Internal Representation of Data Structures

2.1 XML Schema

The formulation of the data representation is important in order to maximize information that can be derived from it. XML Schemas normally have a highly hierarchical tree-like formation. For the purposes of XML Interoperability, data structure is synonymous with schema since a valid XML document (itself a data receptacle) must conform to some XML Schema. Thus its structure must be known — just parse the schema. Structural information-wise, the two are synonymous. Internal representation of the schema is faithful to its tree structure, additionally storing tag data in child nodes of the nodes corresponding to that tag, and attributes likewise in child nodes of the appropriate tag nodes. Hierarchy edges express hierarchy relationships within data domains; Association edges express relationships between elements across data domains. Formally, the representation $\mathcal{X}$ consists of a tuple of components.

$$\mathcal{X} = \langle E_A, D_1, D_2...D_n, s\rangle$$

Where the components are defined as follows:

- E_A is the set of association edges, added by the engine in the course of its execution; as described they express relationships across domains.

  $$E_A \subset \bigcup_{i \neq j} V_i \times V_j$$

- $D_1, D_2, ...$ are domains, comprising a set of vertices V_i and hierarchy edges E_{H_i}

  $$D_i = \langle V_i, E_{H_i}\rangle = \langle\langle V_{T_i}, V_{D_i}, V_{A_i}\rangle, E_{H_i}\rangle$$

  where V_{T_i} represents the set of tags, V_{D_i} the tag data, and V_{A_i} the attributes. The vertices are linked by directed edges in E_{H_i}.

  $$E_{H_i} \subset (V_{T_i} \times V_{D_i}) \cup (V_{D_i} \times V_{A_i})$$

- The scoring function $s : E_A \to \Re$ or $E_A \to [0, 1]$ is described later.

Both association and hierarchy edges are implicitly asymmetric, $(v_1, v_2) \neq (v_2, v_1)$. For hierarchy edges, this is immediate since direction of the edge is encapsulated in the semantic structure of the schema. For association edges, this captures the earlier observation that not all associations need be two-way and that some are manifestly not so.

Adhering to the tree structure of a valid XML specification imposes a stronger condition. Within each domain D_i there must exist a partial ordering $\leq$ such that $(v, v') \in E_{H_i} \Rightarrow v \leq v'$ (and of course $v, v' \in V_i$). For example, a 3-cycle cannot have such a partial ordering described on it and is, therefore, not a valid domain description.

The above constraint is inherent to XML but may actually be relaxed to our advantage. Notably, the X-Map prototype design does not preclude a "flattening" of the hierarchical tree structure with consolidation of identical nodes. The data structure used (a tagged directed graph) is built to handle any arbitrary graph of tagged nodes. The rationale behind potentially recasting the tree is that for particular structures, a flattened graph will be more computationally palatable. A problem with structural analysis on an open tree is the bias it builds against correlating aggregation and generalization conflicts. In non-trivial cases, these conflicts result in completely different tree structures, which confuse attempts to frame resolutions based on hierarchies. Thus, for our purposes, the schema's "tree" is compacted into a more useful representation.

In either event, this structure manages to preserve the structural information of the XML input, while casting it into a form amenable to analysis. Besides elementary heuristics and more complex AI techniques, casting the data structure into a directed graph as described allows us to leverage the body of existing graph theory work on isomorphism of graphs. Finally, making the distinction between graph edges representing hierarchical associations within domains, as opposed to edges representing mapping-type associations across domains (E_A/E_{H_i}), allows us to represent multiple data domains D_1, D_2, D_3... within a single directed graph. This is important because it enables the association engine to leverage knowledge of mappings between two domains in discovering associations linking them with a third.

Association edges use scores to represent degrees of confidence in nominated associations, rather than a simple on-off distinction. Besides being conceptually simple, this idea is significant in incrementally enhancing understanding of the relationship graph. When augmenting information from previous analyses about a speculative relationship, one needs flexibility in the degree to which current knowledge is changed. A regenerative association engine repeatedly iterates through loops of such analyses to further strengthen the scores. As the final step, all that needs to be done is to compare edge scores against a (predetermined) threshold and take appropriate action for edges that cross that threshold. This is in accordance with the notion that one should never discard information, even uncertain information, when trying to establish associations: one never knows when one will need that information again. Confidence level on edges is retained until those edges are proved to not be associations.

Other modes of execution are being explored. Analysis engines can be run "in parallel", and subsequent generated edge scores combined according to a heuristic of choice. The further extension to this concept is to retain separate scores for each of the analysis techniques applied to the schema, and combine and recombine the scores arbitrarily as per valid criteria.

2.2 Scoring Mechanism

Interpreting the scores as degrees of confidence (probabilities) has the advantage of being conceptually consistent and well-defined. Technically, the more accurate term is "confidence level": the edge definitely is either an association or not, and we are describing the probability of seeing a particular set of sample data *given* that an association exists there. Scores are thereby restricted to the closed $[0, 1]$ interval, and we draw on probability theory for ideas on how to manipulate them. When applying multiple *atomic* analysis techniques, the weighted average of returned scores is taken for each edge.

The combination of confidence estimates from independent sources was considered in detail. For example, when using the definite-association-vicinity heuristic described in the next section, what needs to be done for a nomination that lies in the vicinity of two or more known associations? Taking any kind of average in this case is inappropriate since it implies that adding a further measure of certainty sometimes *decreases* the level of confidence in the association. A measure is needed that incorporates information from both sources but still retains all properties desired by the scoring mechanism.

Formally, the requirement is for a probability combination function $\alpha : [0,1]^2 \rightarrow [0,1]$ that satisfies the following conditions:

- Probabilities must lie between 0 and 1, obviously.
  $0 \leq \alpha(x,y) \leq 1$ for $0 \leq x, y \leq 1$
- Order should not be relevant, so α should be both associative and commutative.
  $\alpha(x,y) = \alpha(y,x)$ and $\alpha(x, \alpha(y,z)) = \alpha(\alpha(x,y), z)$
- Adding confidence levels never decreases the level of confidence.
  $\alpha(x,y) \geq \max(x,y)$
- Adding confidence 0 has no effect.
  $\alpha(x,0) = x$
- Adding confidence 1 indicates complete confidence in the mapping, unaffected by other estimations.
  $\alpha(x,1) = 1$

A little inspection shows that any function of the form

$$\alpha(x,y) = \beta^{-1}(\beta(x) + \beta(y))$$

will suffice, where β is a strictly increasing function such that $\beta(0) = 0, \beta(1) = \infty$. For example, using an exponential function like a sigmoid, $\beta(x) = e^x/(1 - e^x)$ gives

$$\alpha(x, y) = (x + y - 2xy)/(1 - xy)$$

However, using the probability notion and regarding the two influencing associations as 'events' each of which can independently affect the edge in question yields the even simpler formula

$$\begin{aligned} \beta(x) &= -\log(1 - x) \\ \alpha(x, y) &= 1 - (1 - x)(1 - y) \\ &= x + y - xy \end{aligned}$$

This simpler function was selected for purposes of the current analysis.

Chapter 3

Means of Analysis

3.0.1 Discovering Relations

The previous section focused on identifying the relations that interest X-Map. This section will focus on how to identify those relations in XML documents, as well as some background preparation, which will then pave way for the discussion of algorithmic strategies in Section 3.1.3 that discovers these relations.

Like COIN's axiomatic approach toward mediating queries [6], the following set defines X-Map's operational "axioms". This approach is feasible due to its emphasis on document structure, which directly leverages specific domain knowledge and will be discussed in Section 3.1.3.

Algorithm Execution Decision

Basically, X-Map performs an optimization step between loading the XML schema into memory and processing it with its algorithms — X-Map pre-processes the schema and tries to compile a processing strategy on the fly, which X-Map then executes.

X-Map presently employs a simple heuristic to determine which of its algorithms to run on input XML schema. In this case, X-Map uses a computationally cheap circularity check, which, as we will shortly discuss in Section 3.1.3, is a good indicator of aggregation conflicts. Other cheap heuristics can be performed at this stage, each relevant to the algorithm it precedes, which will ultimately drive down computational costs while increasing relevant matchings.

However, this is purely an exercise in optimization to finding the right set of algorithms to run, not interoperability, so X-Map may, for simplicity, run all of its heuristics such as the circularity check.

Structural Analysis Background

The major realization underlying X-Map's structural analysis approach is the formulation of the data structure. This is important because it determines the types and number of analyses that can be run.

Due to the highly hierarchical and tree-like structure of XML schemas, the importance to recast this tree into a more computationally palatable form cannot be underestimated. The fundamental problem with structural analysis on a tree is the bias it builds against correlating aggregation and generalization conflicts. In non-trivial cases, these conflicts typically result in completely different tree structures which confound any attempts to frame any resolutions based on hierarchies. Thus, for XML Interoperability purposes, the schema's "tree" must be altered into a better representation.

This realization leads X-Map to propose a non-traditional view of a tree — as a directed graph — for XML Interoperability, as shown in the next chapter in Figure 4-2. Basically, tree nodes map to graph vertices, tree edges map to graph edges, and the edge direction goes from the root node to the child node.

This view transformation dramatically increases the number of interesting properties (of graphs) and also the analysis options one can perform on a tree hierarchy, and as Section 3.1.3 will show, this transformation can be done quickly.

Thus, the following list represents some[1] of the interesting features of a directed graph which X-Map will take advantage of, and they will be discussed in Section 3.1.3. Basically, each represents an embodiment of schematic or semantic conflicts mentioned earlier, and shows how X-Map uses the information.

- Associations in Hierarchies
- Associations in Cycles
- Associations through Contradiction

3.0.2 X-Map's Strategies

As briefly mentioned in Section 3.1, X-Map's strategy employs a number of analysis algorithms on structure and language, aided with specific knowledge of the project's problem domain, to meet and resolve the XML Interoperability problem.

[1]By no means is this list all inclusive. Additional graph features to exploit can easily be the topic of future research.

A key feature of X-Map is its regenerative association engine (see Section 3.1.4), which not only validates known associations but also keeps track of uncertain associations so that they can be ascertained when new future information becomes available to X-Map.

Equally critical to X-Map's strategy is the idea of recasting a hierarchical data structure like XML into a directed graph and leveraging the body of graph theory work on it.

Finally, at all times, human intervention may prove more fruitful in resolving the final outstanding issues after each sub-analysis has finished weeding through the complexity. This proves to be a benefit to both the operator and the algorithm since the algorithm is ultimately not sentient and may require suggestions only a human can make, so the algorithm will go through the complexity which overwhelms most humans to extract the basic conflict in question.

Thus, the following sections will explain the process and rationale behind the graph analyses that X-Map uses along with how the regenerative association engine "does its magic".

How to Collapse a Tree Hierarchy

The process of converting XML into a directed graph and vice versa proves deceptively simple due to existing software for another project at MITRE. The basic concept behind the "Bags" concept is that a hierarchy is simply a collection of nodes whose edges represent some attribute or containment relationship between the appropriate parent and child node. [34]

However, instead of focusing on how the relations stack together to form a tree-like hierarchy, if one focuses on the nodes and the relations they contain or participate, the graph formulation immediately leaps forth. Thus, "XML-2-Bags" embodies this realization to fruition. “XML-2-Bags”, in pseudo-code, simply:

Also, one must realize that in addition to “flattening” the tree into a directed graph, X-Map will keep the hierarchical information around to aid its graph algorithms in determining relations.

Finally, as explained earlier in Section 3.1.2, viewing a tree as a directed graph holds much potential, as the following sections will show.

Discovering Associations in Hierarchies

Discovering associations between elements in a hierarchy can be tricky because the hierarchy is not guaranteed to contain any “meaning” for the associated elements. Fortunately, in this project, it is often the case that hierarchy embeds information about its constituent elements. This is domain knowledge specific to this project that will be exploited.

The typical example (with A, B, C, D, E as elements and arrows pointing in the to part of the relation) would be: A contains B; B contains C; and D contains E. If C and E are related, does that say anything about B and D? What about A and D?

For this project, it often turns out that if C and E are related, then either A or B are related to D. However, even if the relation cannot be immediately drawn from A or B to D, one can still speculate on its existence. Future processes may discover that B and D are related without the presence of C and E, in which case X-Map now has a more complete picture than either processes alone.

Furthermore, if a correlation is drawn both between C and E and an encompassing parent such as A and D, that more than likely sheds more light about the correlation of elements in between. Perhaps the intervening elements in one schema correspond to additional and yet discovered detail for the other schema.

Deriving Associations in Cycles

Other times, an association can be made that is cyclic — that is, there is an association between an encompassing element and the contained element between the schemas. An example of this would be: A contains B; C contains D. A is related to D and C is related to B.

Directed cycles often indicate the presence of aggregation and generalization conflict due to similar "sorts" of information organized or aggregated differently for different data-domains. The resolution of this conflict shows the advantages of the graph over the tree — a tree will stead-fastly maintain such hierarchical conflicts during analysis and frustrate its resolution while a graph does not.

As noted in Section 3.1.3, this situation potentially allows X-Map to speculate on the meaning of intervening elements, too. Perhaps element E represents details from schema 1 that schema 2 does not yet exhibit and can be confirmed in the future through speculation.

Dealing with Conflicting Associations

Clearly, it is possible to derive contradictory associations such as "A implies B and A does not imply B" or one-sided associations such as "A implies B but B does not imply A". The latter situation possibly implies a substitutable relation while the former is slightly thornier. In general, situations like this can be handled by having a persistent body of knowledge that either affirmatively says "yes, A implies B", or "no, A does not imply B". The fully qualified means of building this knowledge is an active area of research.

X-Map builds this body of knowledge through its relations linkbase in a couple of ways.

- Some schemas may be more trusted than others; thus, relations drawn from them may have higher weight to break contradictory associations.
- X-Map can fully speculate on this result and wait for future schemas to resolve this issue via the regenerative association engine.

3.0.3 Regenerative Association Engine

The regenerative association engine embodies a simple logical concept extremely applicable toward XML Interoperability:

> One should *never* throw away information, even uncertain ones, when trying to reason out and associate things — one never knows when one will need that information again.

Hence, The engine's design is surprisingly simple since its tasks basically entail the following:

1. Keep track of the speculative relations and what association algorithm produced each.
2. Rerun the associated algorithm on the speculative relations when new schemas are introduced into X-Map.
3. Record whether a speculative relation's conflict is resolved affirmatively or negatively and act accordingly.

3.1 Heuristics

Discovering associations between elements in a hierarchy can be tricky because the hierarchy is not guaranteed to contain any "meaning" for associated elements. Fortunately, in this project, it is typically the case that hierarchy embeds information about its constituent elements. This is domain knowledge specific to this project that will be exploited. In the case of heuristics, we use fairly simple patterns and functions to derive information from structural associations. Many of these actually depend on prior known mappings — they need to be "bootstrapped" by first running Equivalence Analysis or Data Analysis on the domains in question, or reading off a set of known associations from a Linkbase.

The simplest case involves direct hierarchical associations. Thus if $A \rightarrow B \rightarrow C$ and $D \rightarrow E \rightarrow F$ are pieces of the hierarchy in two distinct domains, and a strong mapping exists between B and E, we can speculate with some (initially low) degree of confidence that C and F are related. We also acquire (slightly higher, but still low) confidence that A and D are related, or their parents.

The next step is noting that the converse is somewhat stronger: If *both* $A - D$ and $C - F$ are related, we have a relatively higher degree of confidence that B and E are related. In some sense, the nearby associations at the same points in the hierarchy reinforce our confidence in the association currently being examined. If the known associations are fairly distant, however, our level of confidence in these two elements being related drops off rapidly.

The score-combining function from the previous section comes in useful in implementing this heuristic. Under this scheme, potential edges are rated based on the proximity of their end nodes to the end nodes of known associations. The more such known associations lie in the vicinity of the speculated edge, the more the edge score gets reinforced. We score nodes based on both proximity and direction with respect to known associations. Thus for instance in the example above, if $B - E$ is a mapping, then $A - D$ should be rated more highly than $A - F$. The latter is still possible however, if there was at some stage ambiguity in which node is the parent and which one the child. Similarly, A's other child – D's other child will be rated more highly, and so on. Associations of the form $A - F$ are not ignored, but acquire lower scores: we have this flexibility in how their relationship is affected.

Other heuristics which could be used include deriving associations in cycles (this would necessitate having run the graph through the *flattening* step described in the previous section: obviously a tree has no cycles). Directed cycles often indicate the presence of aggregation and generalization conflict due to similar "sorts" of information organized or aggregated differently for different data-domains. The resolution of these cycles can frequently indicate to us such associations that we might not have otherwise detected. Finally, we can often usefully "chain" associations. The concept is highly intuitive: given three elements A, B, C in distinct domains, a mapping relationship between A and B, as well as one between B and C, is strongly indicative of a potential relationship between A and C. If the intermediate associations are not definite but speculative, the idea becomes cloudier, but we can still use probability-combination type formulae such as described in the preceding section. What makes this possible is our mode of storing multiple data domains in a single graph representation.

3.2 Graph Theory Analysis

The base problem we're dealing with — finding equivalencies in (unlabeled) graphs — has been fairly deeply analyzed computationally. For certain classes of graphs, solving graph isomorphism is provably NP-complete. There are known randomized algorithms for graph isomorphism, however, (mostly due to Babai and Erdos) that run in $O(n^2)$ time on "almost all" pairs of graphs, i.e., pathological instances for which these algorithms fail are highly unlikely to turn up in real-world situations. The algorithms still need to be tweaked to apply to our problem, for the following reasons:

1. We aren't trying to determine perfect isomorphism, just "likely" links.
2. During analysis we'll already have some of the nodes 'labeled' in the sense that we'll already know how some of them correspond to each other.
3. What makes the problem much harder, however, is the fact that the graphs in our problems are not even perfectly isomorphic, but merely "similar". There will always be some nodes with no correspondencies in the other domain, as well as nodes for which the hierarchy, parenthood structure is not matched by its analogue on the other side.

A logical way to try to attack the problem, aided by definite knowledge of some associations, is to guess the identity of some k vertices in one domain with k in the other, and see what the assumption of these implies. Depending on the intial assignment of k nodes, if we look at powers of the adjacency matrix whose rows are the nodes we have fixed, they give a classification of the columns that will break them up enough to distinguish them. or to make the ones possibly similar to one another much smaller than before.

How efficient will this procedure be? In the worst case, we have to worry about graphs that look like projective planes, for which it would be prohibitively unlikely to choose the correct k nodes in a reasonable number of tries. in the practical case we can possibly use a probabilistic argument averaging over the class of graph problems we derive from XML trees. There is some subjectivity in describing our general data structures in this numerical fashion, but it should not be excessive.

3.3 Other Techniques

My thesis finally focuses on other potential techniques to use for structural analysis, drawn from artificial intelligence. The use of continuous scores opens up a vast number of mathematical operations that can be performed on sets of edges. When we bring into play feedback loops that repeatedly update the scores of edges over several cycles, we make possible the application of neural nets, Bayesian analysis and other techniques which will also take into account existing data and requisite domain knowledge. Essentially, the thesis will provide a set of comparisons of these various analyses to determine which ones work best in the context of the project.

Chapter 4

Heuristics Based Approach

Discovering mappings between elements in an XML environment can be tricky because the hierarchy is not guaranteed to contain any "meaning" for associated elements. Fortunately, in this environment, the hierarchy does usually embed information about its constituent elements. This is domain knowledge specific to our purpose that will be exploited. In the case of heuristics, the engine uses patterns and functions to derive information from structural associations. Many of these actually depend on prior known mappings — they need to be "bootstrapped" by first running Equivalence Analysis and Data Analysis on the domains in question, or reading off a set of known associations from an X-Map Linkbase [34]. Heuristics, like most of the techniques used in structural analysis, serve a complementary rather than a stand-alone role. Figure 4-1 [34] shows an example of where heuristic analysis would fall into the larger picture.

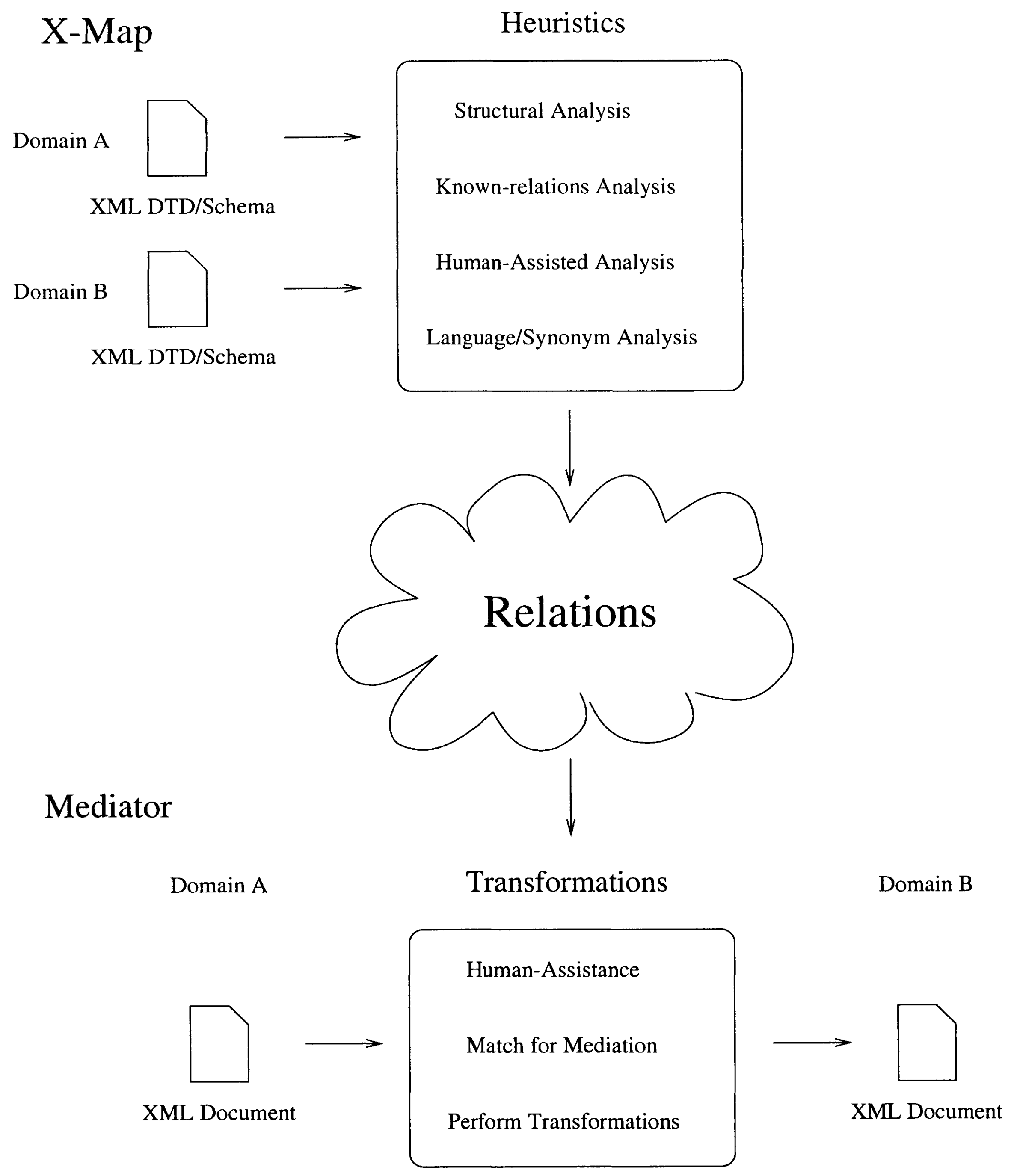


Figure 4-1: Heuristics Application

4.1 Reasoning from Associations

The simplest case involves direct hierarchical associations. Thus if $A \dashrightarrow B \dashrightarrow C$ and $D \dashrightarrow E \dashrightarrow F$ are pieces of the hierarchy in two distinct domains, and a strong mapping exists between B and E, one can speculate with some (perhaps low) degree of confidence that C and F are related. One also acquires (slightly higher, but still low) confidence that A and D are related, or their parents.

Next, note that the converse is somewhat stronger: If *both* $A - D$ and $C - F$ are related, there is a relatively higher degree of confidence that B and E are related. In some sense, the nearby associations at the same points in the hierarchy reinforce confidence in the association currently being examined. If the known associations are fairly distant, however, the level of confidence in these two elements being related drops off rapidly. Based on experiments with sample data, this drop-off was found to be roughly exponential, with best results coming from a discount factor of around 0.65 for the dataset that was analyzed. Of course, the set of actual bests will depend on the precise nature of the data domains being examined. A good avenue for further exploration would be to dynamically increase or decrease this factor based on the characteristics of the domain graphs.

The score-combining function from the previous section turns out to be useful in implementing this heuristic. Under this scheme, potential edges are rated based on the proximity of their end nodes to the end nodes of known associations. The greater the incidence that such known associations lie within the vicinity of the speculated edge, the more the edge score gets reinforced. Nodes are scored based on both proximity and direction with respect to known associations. Thus for instance in the example above, if $B - E$ is a mapping, then $A - D$ should be rated more highly than $A - F$. The latter is still possible however. There might have been ambiguity at some stage in which node is the parent and which one the child. Similarly, A's other child – D's other child will be rated more highly, and so on.

Associations of the form $A - F$ are not ignored, but acquire lower scores; this flexibility is provided in evaluating how their relationship is affected.

4.2 Associations in Cycles

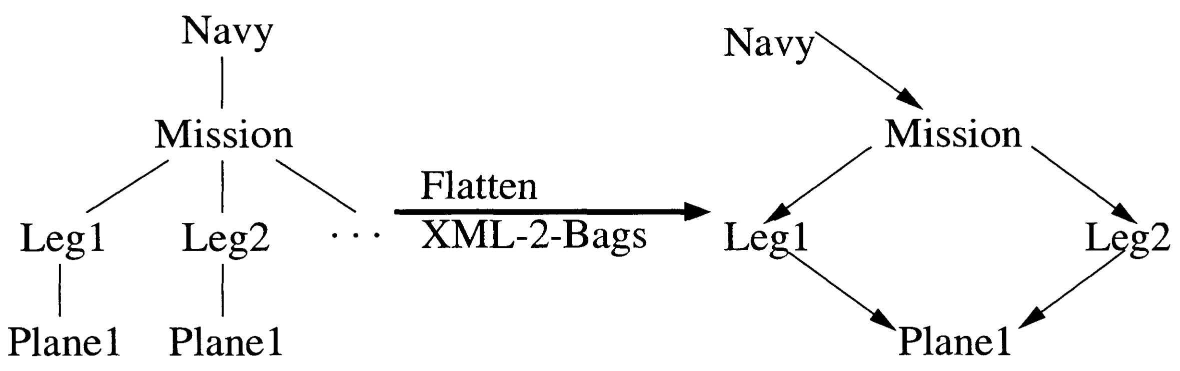


Figure 4-2: Deriving Associations in Cycles

Other methods used include deriving associations in cycles, as in Figure 4-2 [34]. This necessitates having run the graph through the *flattening* step described in the previous section — obviously a tree has no cycles. Directed cycles often indicate the presence of aggregation and generalization conflict due to similar "sorts" of information organized or aggregated differently for different data-domains. The resolution of these cycles can frequently indicate associations that would not otherwise have been detected.

4.3 Chaining Associations across Domains

The concept of "chaining" associations is highly intuitive: given three elements A, B, C in distinct domains, a mapping relationship between A and B, as well as one between B and C, is strongly indicative of a potential relationship between A and C. If intermediate associations are not definite but speculative, the idea becomes cloudier, but one can still use probability-combination type formulae of the type described in the preceding section. What makes this possible is the utilized mode for storing multiple data domains in a single graph representation.

Chapter 5

Discovering Mappings using Graph Theory

5.1 Relation to Graph Problems

In order to identify and analyze semantic interdependencies in a complex XML environment, one powerful approach is to cast the structural analysis problem in terms of graph theory. The underlying problem of finding equivalencies in (unlabeled) graphs has been deeply analyzed computationally [45], and we build upon that research with reference to our particular operational environment. For certain classes of graphs, solving graph isomorphism is provably NP-complete [52, 47]. Subgraph isomorphism, i.e. finding the largest isomorphic subgraphs of given graphs, is also a famous NP-complete problem [46]. There are known randomized algorithms [44, 43] for graph isomorphism that run faster; while these take less time, typically $O(n^2)$, they only work on a subset of the cases. The algorithms still need to be adapted to apply to our problem, for the following reasons:

1. One is not trying to determine perfect isomorphism, just "likely" links.

2. During analysis, some of the nodes will already be labeled in the sense that some of their mutual correspondences are already known.

3. Our problem is much harder, because the graphs in XML problems are not perfectly isomorphic, but merely "similar". There will always be some nodes with no correspondences in the other domain, as well as nodes for which the hierarchy and the parenthood structure are not matched by their analogs on the other side.

We describe a domain (i.e. a graph G) as a set V of vertices and a set E of edges. Given two domains $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, the problem is to find $V_1' \subset V_1$ and $V_2' \subset V_2$ such that the graphs induced by V_1' and V_2' on G_1 and G_2 are isomorphic. Also $|V_1'|$ or $|V_2'|$ is to be as large as possible. The bijection $f : V_1' \rightarrow V_2'$ is simply to be expressed in the association edges: our X-Map structure $\mathcal{X}$ contains E_A such that $f(v_1) = v_2 \Leftrightarrow (v_1, v_2) \in E_A$. For isomorphism, one must also have $(v_1, v_2) \in E_1 \Leftrightarrow (f(v_1), f(v_2)) \in E_2$.

The maximum *bounded* common induced subgraph problem (MAX-CIS) is the same problem with restricted space of input instances; so now G_1, G_2 are constrained to have degree at most B for some constant B [45]. Bounding degrees by a constant factor is a realistic assumption for human-usable XML files, so it is worthwhile to also consider the (perhaps easier) problem of MAX-CIS. Unfortunately, MAX-CIS is also provably NP-complete; [45] gives a reduction to the well-known MAX-CLIQUE problem.

5.2 Solving Isomorphism via Adjacency Matrices

Approaches proposed by other researchers to attack this problem include NAUTY [48], the Graph Matching Toolkit [49], and Combinatorica [50]. Based on our target application, a different approach was implemented as described in the following paragraphs.

A logical way to address the problem, aided by definite knowledge of some associations, is to guess the correspondence of some k vertices in one domain with k in the other, and see what the assumption of these correspondences implies. (Of course corresponding vertices would need to have the same or "nearly the same" degree). Depending on the initial assignment of k nodes, if one looks at powers of the adjacency matrix whose rows are the nodes that have been fixed, these matrices provide a classification of the columns that will help decompose them adequately to distinguish them, or make groups of possibly similar nodes much smaller than before. The higher the power of the adjacency matrix, the more "distanced" the classification is from the original nodes and in the non-identical approximation case, the less reliable it will be.

Evaluating the efficiency of the above procedure is not easy. In the worst case, one needs to deal with graphs that look like projective planes, for which it would be very unlikely to choose the correct k nodes in a reasonable number of tries. In the practical case one could use a probabilistic argument averaging over the class of graph problems derived from XML trees. While there would be some subjectivity in describing our general data structures in this numerical fashion, this would not be excessive as XML graphs are characterized by a restricted structure.

The X-Map prototype currently incorporates an implementation of the algorithm described above.

5.3 Other Avenues

Although not used in the prototype, one alternative strategy was analyzed in detail. Erdös and Gallai's classic extremal function [53] gives the size of the smallest maximum matching, over all graphs with n vertices and m edges. This is the exact lower bound on $\gamma(G)$, the size of the smallest matching that a $O(m+n)$ time greedy matching procedure may find for a given graph G with n vertices and m edges. Thus the greedy procedure is asymptotically optimal: when only n and m are specified, no algorithm can be guaranteed to find a larger matching than the greedy procedure. The greedy procedure is in fact complementary to the augmenting path algorithms described in [29]. The greedy procedure finds a large matching for dense graphs, while augmenting path algorithms are fast for sparse graphs. Well known hybrid algorithms consisting of the greedy procedure followed by an augmenting path algorithm execute faster than the augmenting path algorithm alone [54].

We can prove an exact lower bound on $\gamma(G)$, the size of the smallest matching that a certain $O(m + n)$ time greedy matching procedure may find for a given graph G with n vertices and m edges. The bound is precisely Erdös and Gallai's extremal function that gives the size of the smallest maximum matching, over all graphs with n vertices and m edges. Thus the greedy procedure is optimal in the sense that when only n and m are specified, no algorithm can be guaranteed to find a larger matching than the greedy procedure. The greedy procedure and augmenting path algorithms are seen to be complementary: the greedy procedure finds a large matching for dense graphs, while augmenting path algorithms are fast for sparse graphs. Well known hybrid algorithms consisting of the greedy procedure followed by an augmenting path algorithm are shown to be faster than the augmenting path algorithm alone. The lower bound on $\gamma(G)$ is a stronger version of Erdös and Gallai's result, and so the proof of the lower bound is a new way of proving of Erdös and Gallai's result.

The following procedure is sometimes recommended for finding a matching that is used as an initial matching by a maximum cardinality matching algorithm [29]. Start with the empty matching, and repeat the following step until the graph has no edges: remove all isolated vertices, select a vertex v of minimum degree, select a neighbor w of v that has minimum degree among v's neighbors, add $\{v, w\}$ to the current matching, and remove v and w from the graph. This procedure is referred to in this paper as "the greedy matching procedure" or "the greedy procedure."

In the worst case, the greedy procedure performs poorly. For all $r \geq 3$, a graph D_r of order $4r + 6$ can be constructed such that the greedy procedure finds a matching for D_r that is only about half the size of a maximum matching [32]. This performance is as poor as that of any procedure that finds a maximal matching.

On the other hand, there are classes of graphs for which the greedy procedure always finds a maximum matching [32]. Furthermore, using a straightforward kind of priority queue that has one bucket for each of the n possible vertex degrees, the greedy procedure can be made to run in $O(m + n)$ time and storage for a given graph with n vertices and m edges [55]. The $O(m + n)$ running time is asymptotically faster than the fastest known maximum matching algorithm for general graphs or bipartite graphs [21, 23, 24, 25, 26, 27, 28, 30]. The greedy procedure's success on some graphs, $O(m+n)$ time and storage requirements, low overhead, and simplicity motivate the investigation of its performance.

The matching found by the greedy procedure may depend on how ties are broken. Let $\gamma(G)$ be the size of the smallest matching that can be found for a given graph G by the greedy procedure, i.e., $\gamma(G)$ is the worst case matching size, taken over all possible ways of breaking ties.

We will show that each graph G with n vertices and $m \geq 1$ edges satisfies

$$\gamma(G) \geq \min\left(\left\lfloor n + \frac{1}{2} - \sqrt{n^2 - n - 2m + \frac{9}{4}} \right\rfloor , \left\lfloor \frac{3}{4} + \sqrt{\frac{m}{2} - \frac{7}{16}} \right\rfloor \right). \qquad (5.1)$$

It will become clear that this bound is the best possible — when only n and m are given, no algorithm can be guaranteed to find a matching larger than that found by the greedy procedure.

The simpler but looser bound of $\gamma(G) \geq m/n$ is proved in [55].

The bound in (5.1) can be considered alone, or in conjunction with augmenting path algorithms — the fastest known algorithms for finding a maximum matching. All known worst-case time bounds for augmenting path algorithms are $\omega(m + n)$. It is traditional to use a hybrid algorithm: first, use the greedy procedure (or one like it) to find a matching M in $O(m + n)$ time; then, run an augmenting path algorithm with M as the initial matching. We will see that (5.1) supports the use of such hybrid algorithms. Intuitively, if the input graph is dense, then the greedy procedure finds a large matching, and the augmenting path algorithm needs only a few augmentation phases; if the input graph is sparse, then each augmentation phase is fast.

We can abstract the following technique for solving maximum cardinality matching problems: use one kind of method (perhaps the greedy procedure) for handling dense graphs, and another kind of method (perhaps an augmenting path algorithm) for handling other graphs. It may be interesting to investigate whether existing matching algorithms can be improved upon by explicitly using this technique.

Chapter 6

Artificial Intelligence Based Analysis — Neural Nets

If one looks at the problem as a search space question, there are a number of relevant AI techniques which can be brought into play to handle it. The use of continuous scores opens up a vast number of mathematical operations that can be performed on sets of edges. When one brings into play feedback loops that repeatedly update the scores of edges over several cycles, one can apply neural nets, Bayesian analysis and other technologies which will also take into account existing data and requisite domain knowledge.

6.1 Neural Nets

Neural Networks (NNs) fit into the fourth layer of a top-down algorithm architecture (Figure 6-1). NNs learn from past performance to determine candidates for the current session [57]. In the case of NNs, a description of the subgraph around a potential edge serves as

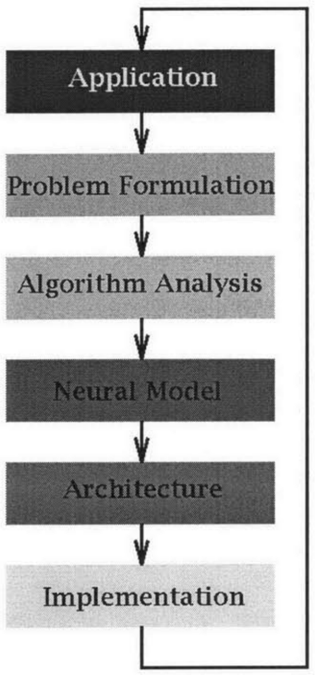


Figure 6-1: Algorithmic Approach

input to the NN. An interesting feature/weakness of techniques of this type is the degree to which they use the "past to predict the future". They need to be trained on test data before being used on fresh domains; this means that if the training data are significantly different from the actual data in target domains, the results posses little relevance. The use of these techniques thus makes some assumptions about the validity of past training data with respect to current resources.

Arguably, the greatest value of neural networks lies in their pattern recognition capabilities. Neural networks have advantages over other artificial intelligence techniques in that they allow continuous, partial and analog representations. This makes them a good choice in recognizing mappings, wherein one needs to exploit complex configurations of features and values.

6.2 NNs in X-Map

Clifton and Li [56] describe neural networks as a bridge across the gap between individual examples and general relationships. Available information from an input database may be used as input data for a self-organizing map algorithm to categorize attributes. This is an unsupervised learning algorithm, but users can specify granularity of categorization by setting the radius of clusters (threshold values). Subsequently, back-propagation is used as the learning algorithm; this is a supervised algorithm, and requires target results to be provided by the user. Unfortunately, constructing target data for training networks by hand is a tedious and time-consuming process, and prone to pitfalls: the network may be biased toward preferring certain kinds of mappings, depending on how the training data is arranged.

The contribution of application-driven neural networks to structural analysis hinges upon three main characteristics:

1. Adaptiveness and self-organization: it offers robust and adaptive processing by adaptively learning and self-organizing.

2. Nonlinear network processing: it enhances the approximation, classification and noise-inmunity capabilities.

3. Parallel processing: it employs a large number of processing cells enhanced by extensive interconnectivity.

Characteristics of schema information, such as attributes, turn out experimentally to be very effective discriminators when using neural nets to determine tag equivalence. The policy of building on XML turns out to be very useful: schema information is always available for a valid document. Furthermore, our scoring mechanism for speculated edges lends itself well to the continuous analog input/output nature of neural networks.

The neural model gets applied in two phases:

Retrieving Phase : The results are either computed in one shot or updated iteratively based on the retrieving dynamics equations. The final neuron values represent the desired output to be retrieved.

Learning Phase : The network learns by adaptively updating the synaptic weights that charcterize the strength of the connections. Weights are updated according to the information extracted from new training patterns. Usually, the optimal weights are obtained by optimizing (minimizing or maximizing) certain "energy" functions. In our case, we use the popular least-squares-error criterion between the teacher value and the actual output value.

6.3 Implementation

The implementation was based on the toolkit JANNT (JAva Neural Network Tool) [63] A supervised network was used. Figure 6-2 [62] schematically illustrates such a network, where the "teacher" is a human who validates the network scores.

Updating of weights in the various layers occurs according to the formula

$$w_{ij}^{(m+1)} = w_{ij}^{(m)} + \Delta w_{ij}^{(m)}$$

where $w^{(m)}$ refers to the node weights at the m'th iteration, and Δ is the correcting factor. The training data consists of many pairs of input-output patterns.

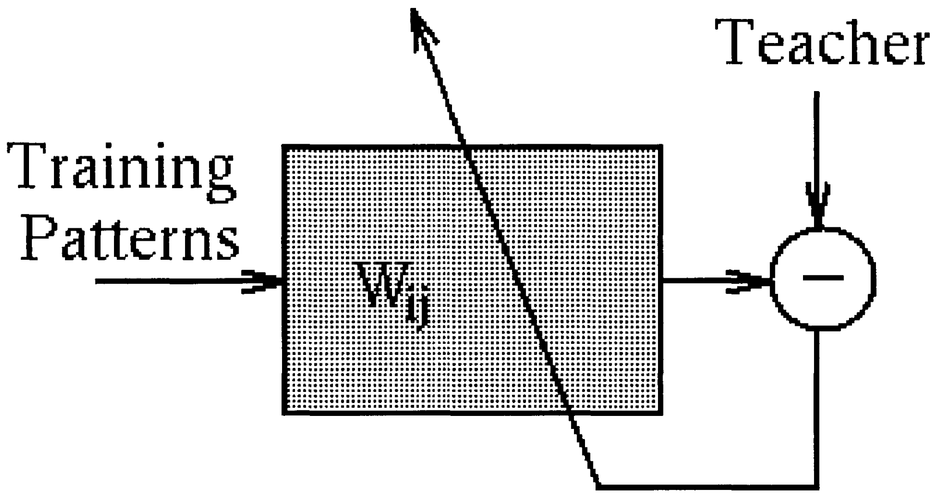


Figure 6-2: Supervised Neural Net

For input vector x, the linear basis function u is the first-order basis function

$$u_i(w, x) = \sum_{j=1}^{n} w_{ij} x_j$$

and the activation function f is the sigmoid

$$f(u_i) = \frac{1}{1 + e^{-u_i/\sigma}}$$

The approximation based algorithm can be viewed as an approximation regression for the data set corresponding to real associations. The training data are given in input/teacher pairs, denoted as $[\mathcal{X}, \mathcal{T}] = \{(x_1, t_1), (x_2, t_2)...(x_M, t_M)\}$, where M is the number of training pairs, and the desired values at the output nodes corresponding to the input $x^{(m)}$ patterns are assigned as teacher's values. The objective of the network training is to find the optimal weights to minimize the error between "target" values and the actual response. The criterion is the minimum-squares error observed.

The model function is a function of inputs and weights: $y = \phi(x, w)$, returning edge probability y. The weight vector w is trained by minimizing the energy function along the gradient descent direction:

$$\Delta w \propto -\frac{dE(x,w)}{dw} = (t - \phi(x,w))\frac{d\phi(x,w)}{dw}$$

For X-Map, a standard feed-forward back-propagation network was used, with two layers, but further efforts could focus on other configurations as well.

Chapter 7

Results

In the preceding sections, we examined three broad categories of approaches to perform structural analysis on XML formatted data between similar but non-identical domains — heuristic based approaches; graph theoretic approaches; and AI based approaches. Having implemented instances of each of these, it was instructive to compare the performance on our prototype application, to see which ones yield promising results. The nature of the problem requires that human validation be involved in evaluating success: specifically, the technique must be executed on data for which a human has already specified which are and which are not viable mappings. An algorithm can then be appraised based on a metric that takes into account both correctly located connections (positive) and false connections incorrectly marked (negative).

7.1 Technique Performances

Comparative results are shown in Table 7.1.

	# actual mappings nominated	# non-mappings nominated	Score % (correct - incorrect)
Total	13	245	
Bootstrapped (already known)	4	0	
Number to be detected)	9	0	
Heuristics	7	10	73%
Adjacency matrices	5	15	49%
Neural networks	5	37	39%

Table 7.1: Performance of Structural Analysis Techniques

Table 7.1 was constructed as follows. Two similar sample data domains were presented to the structural analysis algorithm. The domains were initially compared by a human interpreter, and mappings noted. Then, some of the obvious mappings were bootstrapped into the graph, to simulate prior knowledge discovered by equivalence analysis and data analysis. Clearly, these static analyses could not discover all relevant associations. The algorithm's task was to find previously undiscovered associations.

The score S_i of the i'th technique was computed according to the proportion of associations it got right, minus its proportion of failures:

$$S_i = \frac{c_i}{T - K} - \frac{w_i}{F};$$

- c_i and w_i are the number of correct and incorrect mappings, respectively, nominated by the algorithm.

- T is the total number of correct mappings that existed between the sampled domains, F is the number of "false" mappings (i.e. element pairs that had no relationship), and K is the number of mappings that were known ahead of time (i.e., discovered by some other analysis).

These scores were taken on test domains consisting of sample XML files based on real-world data. One issue which arises in evaluating techniques is that structural analysis is a fairly new approach to entity correlation. There is little comparative information from other researchers regarding alternative approaches.

Overall heuristic approaches performed significantly well, both in determining valid matches and in avoiding fake ones. Not only are these methods straightforward to conceive and implement, they also offer high transparency. It is easy to understand how the internal settings affect final results; this facilitates the modification of parameters to maximize effectiveness of the algorithm for a particular purpose. In contrast, more sophisticated alternatives tended to be "black boxy" in that it is difficult to relate the internals of the process to its final outcome. Heuristic methods showed errors of under 30% on our restricted data sets.

The adjacency matrix implementation was partially transparent, but did not yield as low an error rate. The worst cases arose on sub-schema with a relatively linear structure, or when adjacent nodes in one hierarchy were displaced by having some redundant node between them in another hierarchy. Incidentally, the graph theoretic analysis turned out to require less "bootstrapping" by other analyses than either heuristics or neural nets. The analysis draws more information from the schema structure than from prior known associations across the domains.

Neural networks turned out to be unpredictable and difficult to modulate for our restricted datasets. However, this should not be taken to mean that this option is impractical for the structural analysis problem as a whole. The real power of neural networks and other more elaborate techniques shows up on inputs of greater size and complexity than was reflected in our tests. Since they acquire knowledge cumulatively (rather than recomputing everything on a per-analysis basis) they also show better results on a longer timescale. Due to this characteristic, a neural networks based approach has potential for further investigation.

7.2 Conclusion

In semi-automatically discovering semantic links between elements of heterogeneous domains, such as a cross-internet application, structural analysis of the domains can serve as an invaluable aid. This paper has outlined several of the possible approaches for performing such analysis. The proposed approaches have been incorporated into the X-Map prototype system of an automated association engine. Our current efforts are geared toward performing quantitative comparisons between these analytic techniques, in order to determine which might yield the best results in mediating between non-identical data domains. The objective of such an analysis is to develop an integrated approach that can automatically adopt the optimal approach tailored for the particular set of data, the particular source of the data, and the desired characteristics for the user and the data.

Appendix A

Source Code

```
/*
 * AbbreviationsTable.java
 *
 * Revision History
 *   initial version - 06/21/2000: David Wang
 *   changed version - 06/22/2000: Eddie Byon
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
*/

package org.mitre.XMap;

import org.mitre.XMap.LookupTable;
import org.mitre.XMap.UniqueHashMap;
import org.mitre.XMap.DuplicateException;

import java.util.Iterator;
import java.io.*;

/**
 * An AbbreviationsTable contains information on all possible abbreviations of
 * a given element.  When queried, it basically returns a UniqueHashMap of all
 * possible abbreviations for the queried-for element.  It can potentially be
 * a wrapper on an external source of information.
 *
 * @author David Wang
 * @author Eddie Byon
 * @version 1.0
 * @see org.mitre.XMap.EquivalenceAnalysis
 * @see org.mitre.XMap.LookupTable
 */
public class AbbreviationsTable implements LookupTable {


        /** Abbreviation Table created for EquivalenceAnalysis. */
    private LinkedLookupTable abbrevTable;

    /**
     * Constructs a new table of abbreviations for lookup.
     * Values must be a UniqueHashMap or references to.
     * @since 1.0
     */

    public AbbreviationsTable() {

                try {
                        abbrevTable=new LinkedLookupTable();
                        UniqueHashMap singleLine;
                        Iterator line;
                        FileReader fr = new FileReader("Abbreviations.txt");
                        StreamTokenizer st = new StreamTokenizer(fr);
                        String s, nextEntry;
                        st.eolIsSignificant(true);
                        st.wordChars(33, 127);

                        st.nextToken();
                        while(st.ttype != StreamTokenizer.TT_EOF) {
                                singleLine = new UniqueHashMap();
                                while(st.ttype != StreamTokenizer.TT_EOL) {
                                        singleLine.put(st.sval.toUpperCase(), "");
                                        st.nextToken();
                                }
                                st.nextToken();
                                line = singleLine.keySet().iterator();
                                while(line.hasNext()) {
                                        nextEntry=(String)line.next();
                                        abbrevTable.add(nextEntry.toUpperCase(), singleLine);
                                }
                        }
            fr.close();
                }
                catch (FileNotFoundException e) {}
                catch (IOException e) {}
                catch (DuplicateException e) {
                        //Already in the table.. no need to do anything.
                }
    }

    /**
     * Returns a UniqueHashMap which contains all the
     * abbreviations of key.
     *
     * @param key string whose various abbreviations are searched for
     * @return UniqueHashMap with all abbreviations
     * @exception NotFoundException if no abbreviations are found
     */
    public Object lookFor(Object key)
                throws NotFoundException {
                //BUGBUG: this should do some HashMap lookup if not wrapper
                if (abbrevTable.contains(key))
                        return abbrevTable.get(key);
                return null;
    }
}
```

```
/*
 * AbstractAnalysisAlgorithm.java
 *
 * Revision History
 *   initial version - 06/19/2000: David Wang
 *   2nd version - 9/25/2000: Ashish Mishra
 *   3rd version - 11/03/2000: Eddie Byon
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.Iterator;

import org.mitre.XMap.Equivalence;
import org.mitre.XMap.UniqueHashMap;
import org.mitre.XMap.XMapTDG;

/**
 * This class represents the notion af an abstract analysis algorithm for XMap.
 * All analysis algorithms employed by XMap must extend this class since it
 * provides functionality that XMap expects of its analysis program ''modules''.
 *
 * @author David Wang
 * @see org.mitre.XMap.StructuralAnalysis1
 */
public abstract class AbstractAnalysisAlgorithm {
    /**
     * The key value used to look up the default Equivalence definition for this
     * algorithm.
     */
    public static final String DEFAULT_EQUIVALENCE_KEY = "*";

    /** The lookup table of domains and relevant Equivalence definition. */
    private UniqueHashMap equivalenceDefnTable;

    /** The name of this algorithm implementation. */
    private String algorithmName;

    /**
     * Constructs a new analysis algorithm with no default Equivalence definition.
     * An algorithm should never be created without a default Equivalence definition
     * - this no-arg constructor should be used for subclassing convenience
     * only and subclasses should set a default Equivalence definition by the end
     * of its constructor.
     *
     * @since 1.0
     */
    protected AbstractAnalysisAlgorithm() {
        equivalenceDefnTable = new UniqueHashMap();
    }

    /**
     * Constructs a new analysis algorithm with the given Equivalence definition.
     * <P>
     * The Equivalence definition is required to be not null.
     *
     * @param equivDefn the desired default Equivalence definition
     * @since 1.0
     */
    public AbstractAnalysisAlgorithm(Equivalence equivDefn) {
        this();
        setDefaultEquivalenceDefn(equivDefn);
    }

    /**
     * Associates the Equivalence definition with the given domain for this analysis
     * algorithm.  Domains are restricted to have at most one definition of
     * Equivalence, and this method allows different notions of Equivalence for different
     * domains.
     *
     * @param domain the domain name to be associated
     * @param equivDefn the Equivalence definition to be associated with the domain
     * @since 1.0
     */
    public void setDomainEquivalenceDefn(String domain, Equivalence equivDefn) {
        try {
            equivalenceDefnTable.put(domain, equivDefn);
        } catch (DuplicateException e) {
            //BUGBUG: tell GUI about this substitution, but for now, just do it
            equivalenceDefnTable.remove(domain);
            equivalenceDefnTable.put(domain, equivDefn);
        }
    }

    /**
     * Sets the default Equivalence definition associated with this analysis
     * algorithm.  The lookup key is publicly exposed via DEFAULT_EQUIVALENCE_KEY.
     *
     * @param equivDefn the new default Equals definition
     * @since 1.0
     */
    public void setDefaultEquivalenceDefn(Equivalence equivDefn) {
        setDomainEquivalenceDefn(DEFAULT_EQUIVALENCE_KEY, equivDefn);
    }

    /**
     * Returns the Equivalence definition associated with the given domain.  If the
     * domain is not defined, the default Equivalence definition is returned.
     * <P>
     * The default Equivalence definition is required to be set; if not, the
     * program is halted with an assertion.
     *
     * @param domain the domain whose Equivalence definition is looked up
     * @return the Equivalence definition associated with the given domain
     * @since 1.0
     */
    public Equals getDomainEquivalenceDefn(String domain) {
        try {
            return (Equals)equivalenceDefnTable.get(domain);
        } catch (NotFoundException e) {
            //If the equivalence definition is not found, return the default
            try {
                return (Equals)equivalenceDefnTable.get(DEFAULT_EQUIVALENCE_KEY);
            } catch (NotFoundException ee) {
                                //Since the algorithm should always be constructed with a
                                //default, any error should be fatal
                Assert.assert(false);
                return null; //appease javac
            }
        }
    }

    /**
     * Returns the default Equivalence definition associated with this analysis
     * algorithm.
     *
     * @return the default Equivalence definition of this analysis algorithm
     * @since 1.0
     */
    public Equals getDefaultEquivalenceDefn() {
        return getDomainEquivalenceDefn(DEFAULT_EQUIVALENCE_KEY);
    }

    /**
     * Returns the identifying name of this analysis algorithm.
     *
     * @return the name of this analysis algorithm
     * @since 1.0
     */
    public String getAlgorithmName() {
        return algorithmName;
    }
    /**
     * Sets the identifying name of this analysis algorithm to be
     * newAlgorithmName.
     *
     * @param newAlgorithmName the new name of the algorithm
     * @since 1.0
     */
    public void setAlgorithmName(String newAlgorithmName) {
        algorithmName = newAlgorithmName;
    }

    /**
     * Actually performs the analysis specified by the algorithm.  The
     * directed graph is taken in and the appropriate analysis algorithm is
     * performed, the results of which is returned as an Iterator.  Also,
     * edges are added to the graph representing potential associations.
     *
     * @param graph the directed graph to analyze
     * @return an Iterator of AssociationEdges discovered
     * @since 1.0
     */
    public abstract Iterator analyze(XMapTDG graph);
    //BUGBUG: think about this interface some more.
}
```

```
/*
 * AbstractEquals.java
 *
 * Revision History
 *   initial version - 06/12/2000: David Wang
 *   revised version - 11/03/2000: Eddie Byon
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Equivalence;

/**
 * Defines the common behaviors for all notions of equals.  Namely, a way to
 * identify and describe the notions (so as to help XMap and the GUI).
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.Equals
 * @see org.mitre.XMap.LooseEquals
 */
public abstract class AbstractEA implements Equivalence {
    /** The description for this Equivalence implementation */
    private String equivalenceDescription;

    /**
     * Returns the description for the notion of ''equivalence'' that this
     * implementation uses.
     *
     * @return the description of this implementation's notion of ''equals''
     * @since 1.0
     */
    public String getDescription() {
        return equivalenceDescription;
    }
        /**
     * Sets the description for the notion of ''equals'' that this
     * implementation uses.
     *
         * @param newDescription description for the notion of ''equals.''
     * @return the description of this implementation's notion of ''equals''
     * @since 1.0
     */
    public void setDescription(String newDescription) {
        equivalenceDescription = newDescription;
    }
}
```

```
/*
 * AbstractEquals.java
 *
 * Revision History
 *   initial version - 06/12/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Equals;

/**
 * Defines the common behaviors for all notions of equals.  Namely, a way to
 * identify and describe the notions (so as to help XMap and the GUI).
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.Equals
 * @see org.mitre.XMap.LooseEquals
 */
public abstract class AbstractEquals implements Equals {
    /** The description for this Equals implementation */
    private String equalsDescription;

    /**
     * Returns the description for the notion of ''equals'' that this
     * implementation uses.
     *
     * @return the description of this implementation's notion of ''equals''
     * @since 1.0
     */
    public String getDescription() {
        return equalsDescription;
    }
        /**
     * Sets the description for the notion of ''equals'' that this
     * implementation uses.
     *
         * @param newDescription description for the notion of ''equals.''
     * @return the description of this implementation's notion of ''equals''
     * @since 1.0
     */
    public void setDescription(String newDescription) {
        equalsDescription = newDescription;
    }
}
```

```
/*
 * Assert.java
 *
 * Revision History
 *   initial version - 06/08/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

/**
 * The Assert class simply holds all sorts of static methods useful for
 * asserting a variety of conditions such as not-false, not-null, inclusion and
 * exclusion in range, and equality.  It should never be instantiated.
 *
 * @author David Wang
 * @version 1.0
 */
public class Assert {
    /**
     * Being an utility class of assertions, it makes little sense to
     * instantiate this class and is therefore prohibited.
     */
    private Assert() {
    }

    /**
     * Asserts that b is true.  Otherwise, halt execution.
     *
     * @param b the boolean to test for truth
     * @since 1.0
     */
    public static void assert(boolean b) {
        if (!b) fail("Expected True");
    }

    /**
     * Asserts that o is not null.  Otherwise, halt execution.
     *
     * @param o the Object to test for null-ness
     * @since 1.0
     */
    public static void assert(Object o) {
        if (o == null) fail("Expected non-Null");
    }

    //NOTE: assert for other primitives is != 0??

    /**
     * Asserts that test is inside the inclusive rang bound by low and high,
     * respectively.  Otherwise, halt execution.
     *
     * @param test the int to assert
     * @param low the lower end of the bound to assert, inclusive
     * @param high the upper end of the bound to assert, inclusive
     * @since 1.0
     */
    public static void assertBetween(int test, int low, int high) {
        if (test > high || test < low)
            fail("Expected between [" + low + ", " + high + "]");
    }

    /**
     * Asserts that test is outside the inclusive range bound by low and high,
     * respectively.  Otherwise, halt execution.
     *
     * @param test the int to assert
     * @param low the lower end of the bound to assert, inclusive
     * @param high the upper end of the bound to assert, inclusive
     * @since 1.0
     */
    public static void assertNotBetween(int test, int low, int high) {
        if (test <= high && test >= low)
            fail("Not Expected between [" + low + ", " + high + "]");
    }

    //TODO: assert for other numerical data types

    /**
     * Asserts that test is equal to expected.  Otherwise, halt execution.
     *
     * @param test the int to assert
     * @param expected the int value that is expected
     * @since 1.0
     */
    public static void assertEquals(int test, int expected) {
        if (test != expected)
            fail("Expected " + expected + " == " + test);
    }

    /**
     * Asserts that test is not equal to notExpected.  Otherwise, halt
     * execution.
     *
     * @param test the int to assert
     * @param notExpected the int value that is not expected
     * @since 1.0
     */
    public static void assertNotEquals(int test, int notExpected) {
        if (test == notExpected)
            fail("Not Expected equals " + notExpected + " == " + test);
    }

    /**
     * Asserts that test is equal to expected.  Otherwise, halt execution.
     *
     * @param test the Object to assert
     * @param expected the Object that is expected to be equals to
     * @since 1.0
     */
    public static void assertEquals(Object test, Object expected) {
        if (!test.equals(expected))
            fail("Expected " + expected + " equals " + test);
    }

    /**
     * Asserts that test is not equal to notExpected.  Otherwise, halt
     * execution.
     *
     * @param test the Object to assert
     * @param notExpected the Object that is expected to not be equals to
     * @since 1.0
     */
    public static void assertNotEquals(Object test, Object notExpected) {
        if (test.equals(notExpected))
            fail("Not Expected " + notExpected + " equals " + test);
    }

    /**
     * Called when an assertion fails, fail() throws an Error to halt execution
     * and prints out a stack trace for debugging.
     *
     * @param s the message to be printed in the stack trace
     * @since 1.0
     */
    private static void fail(String s) {
        //System.err.println(s);
        Error e = new Error("Failed Assertion: " + s);
        e.printStackTrace();

        throw e;
    }
}
```

```
/*
 * AssociationEdge.java
 *
 * Revision History
 *   initial version - 9/25/2000: Ashish Mishra
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Edge;
import org.mitre.XMap.XMapNode;
import org.mitre.XMap.TransformFunction;

/**
 * An Association Edge is a subclass of Edge which implements XMap mappings or
 * associations across domains.
 *
 * The score field in an AssociationEdge incorporates the notion of "tentative"
 * certainty about the validity of an edge.  A score is a real number between
 * 0 and 1 which gives our current degree of confidence that there is this
 * significant association between the end nodes.  Different analysis techniques
 * update the score as they uncover more information about the structure of the
 * data domains.  In particular, an edge will get its score updated to 1 or 0 if
 * it is definitively validated or invalidated by a human operator.  A score of
 * 0 is equivalent to not having an edge at all.
 *
 * The equivalenceScore and dataScore fields describe analogous measures of
 * certainty as described by the semantic Equivalence Analysis and Data Analysis.
 * This permits scores to be updated as techniques are run in parallel on the
 * data structures.  At some stage the scores should be combined together to
 * obtain a single measure, which then governs further action to be taken witon
 * the tentative association.
 *
 * @author Ashish Mishra
 * @version 1.0
 * @see org.mitre.XMap.XMapTDG
 */
public class AssociationEdge extends Edge {
    /** The variable which tells us if this edge was output.
     * It's a hack, basically, only used by LinkBase class */
    private boolean outputYet;

    /** The Transformation Function associated with this Edge. */
    private TransformFunction transform;

    /** The type of mapping, as discovered by equals analysis
     * or read in from a linkbase */
    private int mappingType;

    /** The 'level of confidence' that this is in fact a valid association*/
    private double score;
    private double equivalenceScore;
    private double dataScore;

    /*
     * Create an AssociationEdge from newFromNode to newToNode, with specified mapping type
     * and score 1.0 (definite association)
     *
     * @param newFromNode the originating node of this Edge
     * @param newToNode the destinating node of this Edge
     * @param newEdgeTag the new tag value of this Edge
     * @since 1.0
     */
    public AssociationEdge(XMapNode fromNode, XMapNode toNode, int mt) {
            super(fromNode, toNode);
            mappingType=mt;
            score=1.0;
            equivalenceScore=0.0;
            dataScore=0.0;
            transform=null;
}

/*
 * Create an AssociationEdge from newFromNode to newToNode, with specified mapping type
 * and specified score
 *
 * @param newFromNode the originating node of this Edge
 * @param newToNode the destinating node of this Edge
 * @param newEdgeTag the new tag value of this Edge
 * @since 1.0
 */
public AssociationEdge(XMapNode fromNode, XMapNode toNode, int mt, double s) {
            super(fromNode, toNode);
            mappingType = mt;
            score = s;
            equivalenceScore=0.0;
            dataScore=0.0;
            transform=null;
}

/*
 * Returns a boolean telling whether this edge represents a mapping that
 * we're &guot;almost&quot; certain about
 */
public boolean isConfidentMapping() {
            return (score > 0.98);
}

/*
 * Specifies whether this edge has been output to a linkbase file as yet.
 */
public boolean isOutputYet() {
    return outputYet;
}

/*
 * Sets the bit specifying whether this edge has been output to a
 * linkbase file as yet.
 */
public void setOutputYet(boolean opy) {
    outputYet = opy;
}

/** Returns the type of mapping, as discovered by equals analysis
 * or read in from a linkbase */
public int mappingType() {
    return mappingType;
}

/** Sets the type of mapping, as discovered by equals analysis
 * or read in from a linkbase */
public void setMappingType(int mapType) {
            mappingType = mapType;
}

/** Sets the transform function */
public void setTransform(TransformFunction function) {
    transform=function;
}

public TransformFunction getTransform() {
    return transform;
}

public boolean hasTransform() {
    if (transform!=null)
        return true;
    else
        return false;
}

public double getEquivalenceScore() {
    return equivalenceScore;
}
```

```java
    public void setEquivalenceScore(double s) {
        equivalenceScore=s;
    }

    public double getDataScore() {
        return dataScore;
    }

    public void setDataScore(double s) {
        dataScore=s;
        score=s;
    }

    /** Returns the &quot;level of confidence&quot; that this is in fact
     * a valid association*/
    public double getScore() {
        return score;
    }

    public void updateScore() {
        // Arbitrarily increasing by .1???
        // Must check if score is already 1
        // Rename this method
        if (score<1) {
            score+=.1;
            if (score > 0.99)
                score=0.99;
        }
    }

    /** Sets the &quot;level of confidence&quot; that this is in fact
     * a valid association*/
    public void setScore(double s) {
        score = s;
    }

    public void getAllScores() {
        System.out.print(score+ " ");
        System.out.print(equivalenceScore + " ");
        System.out.println(dataScore);
    }

    /** Use for debugging: sends to System.out a two-line synopsis of this edge. */
    public void printEdge() {
        XMapNode fromNode = (XMapNode) getFromNode();
        XMapNode toNode = (XMapNode) getToNode();

        System.out.print(fromNode.getDomain() + ":" + fromNode.getName());
        System.out.print(" --> ");
        System.out.print(toNode.getDomain() + ":" + toNode.getName());
        System.out.print(" - Weight=");
        System.out.println(getScore());
    }
}
```

```
/*
 * Bags2DiGraph.java
 *
 * Revision History
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.domWhiteboard.DomElement;
import org.mitre.xqlParser.GenericNodeList;

import org.mitre.XMap.Node;
import org.mitre.XMap.Edge;
import org.mitre.XMap.EdgeTag;
import org.mitre.XMap.TaggedDiGraph;


/**
 * This class provides interface between Bags & TaggedDiGraph
 *
 */



public class Bags2DiGraph {

        public Bags2DiGraph() {
        }

        public TaggedDiGraph bags2Graph(DomElement rootBag) {
                //...

                TaggedDiGraph tdg = new TaggedDiGraph();
                recursiveB2TDG(rootBag, tdg, null);
                return tdg;
        }


        public void recursiveB2TDG(DomElement rootBag, TaggedDiGraph tdg, Node parent) {
                Node node = new Node();
                node.setBag(rootBag);
                node.setParent(parent);
                try {
                        if (parent != null) {
                                Edge e = new Edge(parent, node);
                                EdgeTag et = new EdgeTag(e, "");
                                tdg.addNodeAndEdge(parent, node, et);
                        } else {
                                tdg.addNode(node);
                        }
                } catch (Exception e) {}

                DomElement childBag;

                GenericNodeList children = (GenericNodeList) rootBag.getChildNodes();
                int numChildren = children.getLength();
                for (int i=0; i<numChildren; i++) {
                        childBag = (DomElement) children.item(i);
                        if (childBag instanceof DomElement) {
                                recursiveB2TDG(childBag, tdg, node);
                        }
                }
        }
}
```

```
/*
 * DTDtoDiGraph.java
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.*;

//import java.io.FileNotFoundException;
import java.io.File;
import java.io.IOException;
//import java.io.Reader;
//import java.io.FileReader;
//import java.net.URL;

// dtd parser code in here
import com.wutka.dtd.*;

/**
 * This class provides interface between DTD & XMapTDG, permitting us
 * to read in DTD files into a directed graph
 *
 * @author Ashish Mishra
 */

public class DTDtoDiGraph {

    public DTDtoDiGraph() {
    }

    public static void dtd2Graph(String fileName, XMapTDG tdg, String domain)
    throws IOException {

        DTDParser dp = new DTDParser(new File(fileName), true);
        // Parse the DTD and ask the parser to guess the root element
        DTD dtd = dp.parse(true);

        recursAdd(dtd.rootElement, tdg, dtd, (XMapNode) null, domain);
    }

    private static void recursAdd(DTDElement elem, XMapTDG tdg,
                                  DTD dtd, XMapNode parent, Object domain) {

        // Recursively add elements to the TDG, going from this
        // element and its attributes to child elements

        XMapNode elnode = new XMapNode(elem.getName(), parent, domain);
        System.out.println(elem.getName());
        try {
            if (parent != null) {
                tdg.addNodeAndEdge(parent, elnode,
                                   new HierarchyEdge(parent, elnode));
            } else {
                tdg.addNode(elnode);
            }

            Enumeration attrs = elem.attributes.elements();
            while (attrs.hasMoreElements()) {
                DTDAttribute attr = (DTDAttribute) attrs.nextElement();
                XMapNode attrnode = new XMapNode(attr.getName(), elnode,
                                                 domain, true);
                tdg.addNodeAndEdge(elnode, attrnode,
                                   new HierarchyEdge(elnode, attrnode));
            }
        } catch (DomainsException e) {
            //Shouldn't happen if parent and this node are same domain
            e.printStackTrace();
        } catch (NoNodeException e) {
            //Shouldn't happen if parent is in the digraph
            e.printStackTrace();
        } catch (DuplicateNodeException e) {
            //Shouldn't happen if this node isn't already in the digraph
            e.printStackTrace();
        } catch (NullPointerException e) {
            System.err.println("This block!!!");

            //Shouldn't happen if this node isn't already in the digraph
            e.printStackTrace();
        }

        //DTDContainer cont = (DTDContainer) elem.getContent();
        DTDItem[] items = ((DTDContainer) elem.getContent()).getItems();
        String itemname;
        DTDElement itemelem;
        for (int i=0; i < items.length; i++) {
            if (items[i] instanceof DTDName) {
                itemname = ((DTDName) items[i]).getValue();
                itemelem = (DTDElement) dtd.elements.get(itemname);
                if (itemelem == null) {
                    itemelem = new DTDElement(itemname);
                    itemelem.attributes = new Hashtable();
                    itemelem.setContent(new DTDSequence());
                }
                recursAdd(itemelem, tdg, dtd, elnode, domain);
            }
        }
    }

}
```

```
/*
 * DataAnalysis.java
 *
 * Revision History
 *   initial version - 12/06/2000: Eddie Byon
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.*;

import org.mitre.XMap.AbstractEA;
import org.mitre.XMap.AbbreviationsTable;
import org.mitre.XMap.MeasurementsTable;
import org.mitre.XMap.IsLikeTable;
import org.mitre.XMap.AcronymsTable;

import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.AssociationEdge;

/**
 * This class implements the ''loose'' notion of equivalence useful for prototyping
 * fuselets.  That is, ''two entities are equivalent when they express essentially
 * the same physical object or concept and differ only by measurement units,
 * mathematical precision, acronyms and/or abbreviations, or other shortening
 * or lengthening of its labelling identifier by concatentation''.  This class focuses
 * on data analysis, that is, comparing data of two entities to see whether or not
 * an association exists.
 *
 * @author Eddie Byon
 * @version 1.0
 */

public class DataAnalysis {
    /**
     * Constructs the default notion of ''loose'' equivalence.
     *
     * @since 1.0
     */

    /** The Data is of type TITLE (10 words or less approximately) **/

    public static final String TITLE = "TITLE";

    /** The Data is of type DESCRIPTION (50 words or less approximately) **/

    public static final String DESCRIPTION = "DESCRIPTION";

    /** The Data is of type SUMMARY (At least 50 words approximately) **/

    public static final String SUMMARY = "SUMMARY";

    public DataAnalysis() {}

    /**
     * Returns a weight denoting how equal two types of data are equal.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @param relationsGraph directed graph in which the nodes are compared
     * @return a weight which will be used
     * @see org.mitre.XMap.EquivalenceAnalysis
     * @since 1.0
     */
    public double data(XMapNode firstNode, XMapNode secondNode, XMapTDG tdg) throws NoNodeExcepti
on {
```

```
String firstDataSetType, secondDataSetType;
ArrayList firstDataSet = firstNode.getData();
ArrayList secondDataSet = secondNode.getData();
ListIterator tempDataSet = firstDataSet.listIterator();
String tempString = "";
double sum=0;

// Get average data entry length for the first Node.

while (tempDataSet.hasNext()) {
    sum+=((String)tempDataSet.next()).length();
}
int firstDataSetAverage = (int)sum/firstDataSet.size();

// Get average data entry length for the second Node.

tempDataSet = secondDataSet.listIterator(); sum=0;
while (tempDataSet.hasNext()) {
    sum+=((String)tempDataSet.next()).length();
}
int secondDataSetAverage = (int)sum/secondDataSet.size();

// Break down into categories -> TITLE, DESCRIPTION, SUMMARY

if (firstDataSetAverage <=60) {
    firstDataSetType = TITLE;
} else if (firstDataSetAverage >=60 || firstDataSetAverage <=250) {
    firstDataSetType = DESCRIPTION;
} else
    firstDataSetType = SUMMARY;

if (secondDataSetAverage <=60) {
    secondDataSetType = TITLE;
} else if (secondDataSetAverage >=60 || secondDataSetAverage <=250) {
    secondDataSetType = DESCRIPTION;
} else
    secondDataSetType = SUMMARY;

// Grab the first instances of the two data arrays.

// START LOOP HERE
int m;
sum=0;

ListIterator firstIterator = firstDataSet.listIterator();
ListIterator secondIterator = secondDataSet.listIterator();
String firstData = "", secondData="";



for (m=0; m<3;m++) {
    if (firstIterator.hasNext()) {
        firstData = (String)firstIterator.next();
    }

    if (secondIterator.hasNext()) {
        secondData = (String)secondIterator.next();
    }
    //                  if (firstNode.getName().equals("format")) {
    //  System.out.print(firstData);
    //  System.out.print(" ");
    //}
    //                          System.out.print(firstData);
    //                          System.out.print(secondData);
    //                          System.out.print("  ");

    int i=0;
    tempString = "";

    if (firstDataSetType == TITLE && secondDataSetType == TITLE) {

        // Store all the words of data into firstWords
        // Break everything down to words, and then begin separate methods
        // as tests.
```

```
            ArrayList firstWords = new ArrayList();
            while(firstData.length()!=i) {
                if (firstData.charAt(i)!=' ')
                    tempString=tempString+firstData.charAt(i);
                else {
                    firstWords.add(tempString);
                    tempString="";
                }
                i++;
            }
            firstWords.add(tempString);

            // Store all the words of data into seocondWords from other domain
            i=0; tempString=""; ArrayList secondWords = new ArrayList();
            while(secondData.length()!=i) {
                if (secondData.charAt(i)!=' ')
                    tempString=tempString+secondData.charAt(i);
                else {
                    secondWords.add(tempString);
                    tempString="";
                }
                i++;
            }
            secondWords.add(tempString);

            // All the words are in the two array lists.  Just create methods
            // and pass the lists on to those methods.  Develop some type of
            // scoring to assign points to the data to see how closely they
            // relate.

            double test = check1(firstWords, secondWords);

            // One word String - "s - s"

            if (test==1.0) {
                test=check2((String)firstWords.get(0), (String)secondWords.get(0));
                test+=check4((String)firstWords.get(0), (String)secondWords.get(0));
            }

            // One word number, like "d - d"

            else if (test==0.9) {
                test=check3((String)firstWords.get(0), (String)secondWords.get(0));
            }

            // multiple words, but same amount of words and types, like "dss - dss"
            else if (test==0.8) {
            }
            // partial match, different number of words, but same pattern. "dss - ss" or "sds
 - sds"
            else if (test==0.7) {
                // do nothing
            }
            //  System.out.print(test);
            //                              System.out.print(" ");
            sum+=test;
        }
        // For now, if any data is longer than 60 chars (10 words), we will dismiss.
        // However, if they are in the same range, we return different values.  This
        // information could help determining associations.

        else if (firstDataSetType == DESCRIPTION && secondDataSetType == DESCRIPTION)
            sum+=.25;
        else if (firstDataSetType == SUMMARY && secondDataSetType == SUMMARY)
            sum+=.125; // Data is of type SUMMARY (over 250 chars)
        else
            sum+= 0.0; // We don't consider cases across types (i.e. TITLE v. SUMMARY)
    }
    return (sum/3);
  }
```

```
    /**
     * First in a series of tests which compares data.  This test checks for
     * data type similarities, as in, if one form of data is represented by
     * integers, and the other is represented by integers as well, then we
     * have a match.
     *
     * @param firstWords ArrayList of data from first domain.
     * @param secondWords ArrayList of data from second domain.
     * @return weight which represents the relationship between data.
     *
     * @see org.mitre.XMap.EquivalenceAnalysis
     * @since 1.0
     */

    public double check1(ArrayList firstWords, ArrayList secondWords) {
        int i=0, j=0, k=0;
        int numberOfWords = firstWords.size();
        int numberOfWords2 = secondWords.size();
        String firstType="";
        char[] temp;
        char[] temp1=null;
        String temp2="";

        // Break the first entry into components.  For example, "Eddie K. Byon"
        // would become "sss" while "16 candles" would become, "ds".

        for(i=0;i<numberOfWords;i++) {
            try {
                temp2=((String)firstWords.get(i));
                //temp=((String)firstWords.get(i)).toCharArray();
                //System.out.print((String)firstWords.get(i));
                //System.out.print(" --- ");
                //temp1 = new char[60];
                //for(j=0;j<temp.length;j++) {
                //    if (temp[j]!=',' || temp[j]!='$') {
                //      temp1[k]=temp[j];
                //      k++;
                //    }
                //}
                //for(j=0;j<k;j++) {
                //    System.out.print(temp1[j]);
                //}
                //
                //temp2=temp1.toString();
                Double myDouble = new Double(temp2);
                firstType+="d";
                k=0;
            }
            catch (NumberFormatException e) {
                firstType+="s";
            }
        }

        // Break the second entry into components.

        i=0; j=0; k=0; String secondType="";
        for(i=0;i<numberOfWords2;i++) {
            try {
                temp2=((String)secondWords.get(i));
                //temp=((String)secondWords.get(i)).toCharArray();
                //temp1 = new char[60];
                //for(j=0;j<temp.length;j++) {
                //    if (temp[j]!=',' || temp[j]!='$'){
                //      temp1[k]=temp[j];
                //      k++;
                //    }
                //}
                //temp2=temp1.toString();
                Double myDouble = new Double(temp2);
                secondType+="d";
                k=0;
            }
            catch (NumberFormatException e) {
                secondType+="s";
            }
        }

        //Uncomment this if you want to see the relationship.
        //      System.out.print(firstType);
        //      System.out.print(" --- ");
        //      System.out.print(secondType);

        if (firstType.equals(secondType)) {
            if (firstType.equals("s"))
                return 1.0; // One word string
            else if (firstType.equals("d"))
                return 0.9; // One word numbers
            else
                return 0.8; // Exact match, more than one word.
        }
        else if ((firstType.endsWith(secondType) || secondType.endsWith(firstType) || firstType.s
tartsWith(secondType) || secondType.startsWith(firstType)) && firstType!="" && secondType!="")
            return 0.7; // This represents matches such as "dss - ss", or "ssd -- sd".
        else
            return 0;  // Cases we could care less about such as "ddd - sss" or "sdsds --- dsssd"
    }


    /**
     * Second in a series of tests which compares data.  This test is given
     * data which contain one word.  The one word is then analyzed character
     * by character, and comparisons are made.
     *
     * @param firstWord data from first domain.
     * @param secondWords data from second domain.
     * @return 1 if it's an exact match, .5 for partial match, 0 for no match.
     *
     * @see org.mitre.XMap.EquivalenceAnalysis
     * @since 1.0
     */

    public double check2(String firstWord, String secondWord) {
        char[] firstString = firstWord.toCharArray();
        char[] secondString = secondWord.toCharArray();
        int i=0;
        String firstType="";
        String secondType="";
        for(i=0; i<firstString.length;i++) {
            try {
                Integer myInteger = new Integer(""+firstString[i]);
                firstType+="i";
            }
            catch (NumberFormatException e) {
                firstType+="s";
            }
        }
        for(i=0; i<secondString.length;i++) {
            try {
                Integer myInteger = new Integer(""+secondString[i]);
                secondType+="i";
            }
            catch (NumberFormatException e) {
                secondType+="s";
            }
        }
        //      System.out.print("    ");
        //      System.out.print(firstType);
        //      System.out.print(" --- ");
        //      System.out.print(secondType);
        if (firstType.equals(secondType))
            if (firstType.indexOf("i")!=-1)
                return 0.8;
            else
                return 0.5;
        else if (firstType.endsWith(secondType) || secondType.endsWith(firstType) || firstType.st
artsWith(secondType) || secondType.startsWith(firstType))
            return 0.0;
        else
```

```
            return 0;
    }

    /**
     * Returns 0.9 if two doubles are within 2 orders of magnitude of each
     * other, 0 if they exceed this limit.
     *
     * @param firstDouble double from one domain to compare
     * @param secondDouble double from another domain to compare
     * @return 0.9 if within 2 orders of magnitude, 0 otherwise.
     * @since 1.0
     */

    public double check3(String firstDouble, String secondDouble)  {
        double sum=0;
        if ((firstDouble.indexOf(".")==-1 && secondDouble.indexOf(".")==-1) || (firstDouble.index
Of(".")!=-1 && secondDouble.indexOf(".")!=-1)) {
            sum=.8;
            if (check4(firstDouble, secondDouble)==0.01)
                sum+=.1;
        }
        else
            sum=.7;

        double double1=Double.parseDouble(firstDouble);
        double double2=Double.parseDouble(secondDouble);

        if (double1==0) {
            if (double2<=10 && double2>=-10)
                return sum;
            else
                return 0.5;
        }
        if (double2==0) {
            if (double1<=10 && double1>=-10)
                return sum;
            else
                return 0.5;
        }
        double typeRatio = double1/double2;
        if (typeRatio<=1) {
            typeRatio = 1/typeRatio;
        }
        if ((typeRatio<=100) && (typeRatio>=0)) {
            return sum;
        }
        return 0.5;
    }
    /**
     * Checks to see if two strings are the same length.
     *
     * @param firstString string from one domain
     * @param secondString string from another domain to compare
     * @return .01 if they're of equal length, or 0 otherwise.
     * @since 1.0
     */

    public double check4(String firstString, String secondString)  {
        if (firstString.length()==secondString.length())
            return .01;
        else
            return 0;
    }
}
```

```
package org.mitre.XMap;

/**
 * Thrown to indicate that the requested node is not found.
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 */
class DifferentDomainsException extends Exception {
        /**
         * Constructs a DifferentDomainsException with no detail message.
         */
        public DifferentDomainsException() {
                super();
        }

        /**
         * Constructs a DifferentDomainsException with the specified detail message.
         *
         * @param s the detail message
         */
        public DifferentDomainsException(String s) {
                super(s);
        }
}
```

```java
/*
 * Edge.java
 *
 * Revision History
 *       initial version - 06/08/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.EdgeTag;
import org.mitre.XMap.Node;

/**
 * An Edge is a data abstraction object that represents a directed edge.  It
 * has a single ''from'' and ''to'' node and optionally have other associated
 * information (its tag).
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 */
public class Edge {
    /** The originating node of this Edge. */
    private Node fromNode;

    /** The destinating node of this Edge. */
    private Node toNode;

    /** The tag associated with this Edge. */
    private EdgeTag edgeTag;

    /**
     * Create an Edge from newFromNode to newToNode with null for its tag value.
     *
     * @param newFromNode the originating node of this Edge
     * @param newToNode the destinating node of this Edge
     * @since 1.0
     */
    public Edge(Node newFromNode, Node newToNode) {
        this(newFromNode, newToNode, null);
    }

    /**
     * Create an Edge from newFromNode to newToNode with newEdgeTage as its
     * tag value.
     *
     * @param newFromNode the originating node of this Edge
     * @param newToNode the destinating node of this Edge
     * @param newEdgeTag the new tag value of this Edge
     * @since 1.0
     */
    public Edge(Node newFromNode, Node newToNode, EdgeTag newEdgeTag) {
        setFromNode(newFromNode);
        setToNode(newToNode);
        setEdgeTag(newEdgeTag);
    }

    /**
     * Returns the originating Node of this Edge.
     *
     * @return the originating Node of this Edge
     * @since 1.0
     */
    public Node getFromNode() {
        return fromNode;
    }

    /**
     * Returns the destinating Node of this Edge.
     *
     * @return the destinating Node of this Edge
     * @since 1.0
     */
    public Node getToNode() {
        return toNode;
    }

    /**
     * Returns the tag associated with this Edge.
     *
     * @return the tag of this Edge
     * @since 1.0
     */
    public EdgeTag getEdgeTag() {
        return edgeTag;
    }

    //BUGBUG: Do we need the set* functionality exposed as public?
    /**
     * Replaces the value of this Edge's originating node with newFromNode.
     *
     * @param newFromNode the new originating node of this Edge
     * @since 1.0
     */

    protected void setFromNode(Node newFromNode) {
        fromNode = newFromNode;
    }

    /**
     * Replaces the value of this Edge's destinating node with newToNode.
     *
     * @param newToNode the new destinating node of this Edge
     * @since 1.0
     */

    protected void setToNode(Node newToNode) {
        toNode = newToNode;
    }

    /**
     * Replaces the value of this Edge's tag with newEdgeTag.
     *
     * @param newEdgeTag the new tag of this Edge
     * @since 1.0
     */
    protected void setEdgeTag(EdgeTag newEdgeTag) {
        edgeTag = newEdgeTag;
    }
}
```

```
/*
 * EdgeTag.java
 *
 * Revision History
 *       initial version - 06/07/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Tag;

/**
 * An EdgeTag is an type of Tag whose value is constrained to be a String.
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 */
public class EdgeTag extends Tag {
    /**
     * Use EdgeTag(Object, String) instead.     It is not desirable to instantiate
     * an EdgeTag without an owner and tag data.
     *
     * @exception Error can't use this constructor
     * @see #EdgeTag(Object, String)
     */
    protected EdgeTag() {
        super();
    }

    /**
     * Creates an EdgeTag with newEdgeOwner as its owner and newEdgeTagData as
     * its tag value string.
     *
     * @param newOwner the new owner of this EdgeTag
     * @param newEdgeTagData the new tag value string of this EdgeTag
     * @since 1.0
     */
    public EdgeTag(Object newEdgeOwner, String newEdgeTagData) {
        super(newEdgeOwner, newEdgeTagData);
        //setOwner(newEdgeOwner);
        //setTagData(newEdgeTagData);
    }

    /**
     * Replaces the value of this EdgeTag with newEdgeTagData.
     *
     * @param newEdgeTagData the new tag value string of this EdgeTag
     * @since 1.0
     */
    public void setTagString(String newEdgeTagData) {
        setTagData(newEdgeTagData);
    }

    /**
     * Returns the value of this EdgeTag.
     *
     * @return the tag value string of this EdgeTag
     * @since 1.0
     */
    public String getTagString() {
        return (String)getTagData();
    }
}
```

```
/*
 * Equals.java
 *
 * Revision History
 *   initial version - 06/12/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Node;
import org.mitre.XMap.TaggedDiGraph;

/**
 * This interface defines the notion of ''equals'' used for semantic matching
 * between schemas.  It can be implemented using whatever algorithm(s) do the
 * job the best.
 * <P>
 * Essentially, a particular instance of an analysis algorithm will be armed
 * with its appropriate implementation(s) of this Equals interface for the data
 * domains involved (so different data domains may have different notions of
 * what equals means, and the analysis algorithm can deal with it).  The
 * algorithm will also have access to the TaggedDiGraph which represents the
 * linkages between the data domains, along with the Node(s) that are being
 * compared for ''equals''.
 * <P>
 * Thus, the analysis algorithm simply uses Equals as a part of its association
 * logic.
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 * @see org.mitre.XMap.AbstractAnalysisAlgorithm
 */
public interface Equals {
    /** The association is exact. */
    public static final int EQ_TYPE_EXACT     = 0x00000000;

    /** The association differs by numerical precision. */
    public static final int EQ_TYPE_PRECISION = 0x00000001;

    /** The association differs by units (English vs Metric, etc). */
    public static final int EQ_TYPE_UNITS     = 0x00000002;

    /** The association say that one is an acronym of the other. */
    public static final int EQ_TYPE_ACRONYM   = 0x00000004;

    /** The association differs only by known abbreviations. */
    public static final int EQ_TYPE_ABBREV    = 0x00000008;

    /** The association differs through arbitrary concatentation. */
    public static final int EQ_TYPE_CONCAT    = 0x00000010;

    /** The association is uncertain, but not none (speculate). */
    public static final int EQ_TYPE_ISLIKE    = 0x00000020;

    /** There is no association. */
    public static final int EQ_TYPE_NOT_EQUIV = 0xF0000000;

    /** There is an pre-determined (de facto stated) association. */
    public static final int EQ_TYPE_NORMAL    = 0x8000000;

    /**
     * Returns an integer that denotes the type of equivalence between
     * firstNode and secondNode given the context of relationGraph.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @param relationsGraph directed graph in which the nodes are compared
     * @return an integer that denotes the type of equivalence reached
     * @since 1.0
     */
    public double sEquals(XMapNode firstNode, XMapNode secondNode, XMapTDG relationsGraph)
        throws NoNodeException;

    /**
     * Returns the description for the notion of ''equals'' that this
     * implementation uses.
     *
     * @return the description of this implementation's notion of ''equals''
     * @since 1.0
     */
    public String getDescription();

    /**
     * Sets the description for the notion of ''equals'' that this
     * implementation uses.
     *
     * @param newDescription the new description of this implementation
     * @since 1.0
     */
    public void setDescription(String newDescription);
}
```

```
/*
 * Equals.java
 *
 * Revision History
 *   initial version - 06/12/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Node;


/**
 * This interface defines the notion of ''equivalence'' used for semantic matching
 * between schemas.  It can be implemented using whatever algorithm(s) do the
 * job the best.
 * <P>
 * Essentially, a particular instance of an analysis algorithm will be armed
 * with its appropriate implementation(s) of this Equivalence interface for the data
 * domains involved (so different data domains may have different notions of
 * what equals means, and the analysis algorithm can deal with it).  The
 * algorithm will also have access to the TaggedDiGraph which represents the
 * linkages between the data domains, along with the Node(s) that are being
 * compared for ''equivalence''.
 * <P>
 * Thus, the analysis algorithm simply uses Equivalence as a part of its association
 * logic.
 *
 * @author David Wang
 * @author Eddie Byon
 * @author Ashish Mishra
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 * @see org.mitre.XMap.AbstractAnalysisAlgorithm
 */
public interface Equivalence {
    /** The association is exact. */
    public static final int EQ_TYPE_EXACT     = 0x00000000;

    /** The association differs by numerical precision. */
    public static final int EQ_TYPE_PRECISION = 0x00000001;

    /** The association differs by units (English vs Metric, etc). */
    public static final int EQ_TYPE_UNITS     = 0x00000002;

    /** The association say that one is an acronym of the other. */
    public static final int EQ_TYPE_ACRONYM   = 0x00000004;

    /** The association differs only by known abbreviations. */
    public static final int EQ_TYPE_ABBREV    = 0x00000008;

    /** The association differs through arbitrary concatentation. */
    public static final int EQ_TYPE_CONCAT    = 0x00000010;

    /** The association is uncertain, but not none (speculate). */
    public static final int EQ_TYPE_ISLIKE    = 0x00000020;

    /** There is no association. */
    public static final int EQ_TYPE_NOT_EQUIV = 0xF0000000;

    /** There is an pre-determined (de facto stated) association. */
    public static final int EQ_TYPE_NORMAL    = 0x8000000;

    /**
     * Returns an integer that denotes the type of equivalence between
     * firstNode and secondNode given the context of relationGraph.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @param relationsGraph directed graph in which the nodes are compared
     * @return an integer that denotes the type of equivalence reached
     * @since 1.0
     */
    public double equivalence(XMapNode firstNode, XMapNode secondNode, XMapTDG relationsGraph)
        throws NoNodeException;

    /**
     * Returns the description for the notion of ''equivalence'' that this
     * implementation uses.
     *
     * @return the description of this implementation's notion of ''equivalence''
     * @since 1.0
     */
    public String getDescription();

    /**
     * Sets the description for the notion of ''equivalence'' that this
     * implementation uses.
     *
     * @param newDescription the new description of this implementation
     * @since 1.0
     */
    public void setDescription(String newDescription);
}
```

```
/*
 * EquivalenceAnalysis.java
 *
 * Revision History
 *   initial version - 06/12/2000: David Wang
 *   changed version - 06/22/2000: Eddie Byon
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.ArrayList;
import java.util.Iterator;

import org.mitre.XMap.AbstractEA;
import org.mitre.XMap.AbbreviationsTable;
import org.mitre.XMap.MeasurementsTable;
import org.mitre.XMap.IsLikeTable;
import org.mitre.XMap.AcronymsTable;

import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.AssociationEdge;

/**
 * This class implements the ''loose'' notion of equivalence useful for prototyping
 * fuselets.  That is, ''two entities are equivalent when they express essentially
 * the same physical object or concept and differ only by measurement units,
 * mathematical precision, acronyms and/or abbreviations, or other shortening
 * or lengthening of its labelling identifier by concatentation''.
 *
 * @author David Wang
 * @author Eddie Byon
 * @version 1.0
 * @see org.mitre.XMap.StructuralAnalysis1
 */
public class EquivalenceAnalysis extends AbstractEA {
    /**
     * Constructs the default notion of ''loose'' equivalence.
     *
     * @since 1.0
     */

    //An acronym is an abbreviation, so we have concluded that the
    //acronymsTable class can just be implemented using the
    //AbbreviationsTable class.


        /** Measurements Table used in Equals Analysis */
    private MeasurementsTable measurementTable;

        /** Abbreviations Table used in Equals Analysis */
    private AbbreviationsTable abbreviationTable;

        /** IsLike Table used in Equals Analysis */
    private IsLikeTable isLikeTable;

    /** The ratio for the Precision Test.  */
    public static final double RATIO    = 100;

    /** Weight used to weigh string manipulations.  */
    public static final double WEIGHT    = 0.5;

        public static int subWeight = 0;

        /**
         * Constructs new class of EquivalenceAnalysis and lookup tables.
         * @since 1.0
         */

    public EquivalenceAnalysis() {
```

```
                super();
                setDescription("Equivalence Analysis");
                //acronymTable = new AcronymsTable();
                measurementTable = new MeasurementsTable();
                abbreviationTable = new AbbreviationsTable();
                isLikeTable = new IsLikeTable();
    }

    /**
         *
     * Returns a weight that denotes the equivalence between firstNode
         * and secondNode given the context of relationGraph.  The weighted
         * values are determined by using lookup tables and string
         * manipulation.  Note that type of equivalence can be created and
         * stored, however, we will wait til the next version to implement this.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @param relationsGraph directed graph in which the nodes are compared
     * @return a weight which will be used
     * @see org.mitre.XMap.Equals
     * @since 1.0
         */

    public double equivalence(XMapNode firstNode, XMapNode secondNode, XMapTDG tdg) throws NoNode
Exception {
                //INITIAL TEST: EXACTNESS
                if (firstNode.getName().toLowerCase().equals(secondNode.getName().toLowerCase()))
{
                        AssociationEdge newAssociation = new AssociationEdge(firstNode, secondNod
e, 1, .99);
                        newAssociation.setEquivalenceScore(0.99);
                        try {
                                tdg.addEdge(firstNode, secondNode, newAssociation);
                        }
                        catch (DuplicateEdgeException e) {
                                System.out.println("Edge already exists.");
                        }
                        return 0.99;
                }
                //END OF INITIAL TEST

                double total = acronymEquivalence(firstNode, secondNode) + abbreviationsEquivalen
ce(firstNode, secondNode);
                if (total>=1) {
                        AssociationEdge newAssociation = new AssociationEdge(firstNode, secondNod
e, 4, 0.99);
                        newAssociation.setEquivalenceScore(0.99);
                        try {
                                tdg.addEdge(firstNode, secondNode, newAssociation);
                        }
                        catch (DuplicateEdgeException e) {
                        }
                        return (int)total;
                }
                else {
                        if ((total<1) && (total>0)) {
                                AssociationEdge newAssociation = new AssociationEdge(firstNode, s
econdNode, subWeight, total);
                                newAssociation.setEquivalenceScore(total);
                                subWeight=0;
                                try {
                                        tdg.addEdge(firstNode, secondNode, newAssociation);
                                }
                                catch (DuplicateEdgeException e) {
                                        System.out.println("This shouldn't happen");
                                }
                        }
                        return EQ_TYPE_NOT_EQUIV;
                }
    }

    /**
     * Returns an integer that denotes if the type of equivalence between
     * firstNode and secondNode is that of an Abbreviation.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @return an integer that denotes if it is an Abbreviation equivalence
     * @see org.mitre.XMap.Equals
     * @since 1.0
     */

    public int abbreviationsEquivalence(XMapNode firstNode, XMapNode secondNode) {

                UniqueHashMap mapofAbbreviations =
                        (UniqueHashMap)abbreviationTable.lookFor(firstNode.getName().toUpperCase(
));
                if (mapofAbbreviations!=null) {
                        if (mapofAbbreviations.containsKey(secondNode.getName().toUpperCase()))
                                return EQ_TYPE_ABBREV;
                }
                return 0;
    }

    /**
     * Returns an integer that denotes if the type of equivalence between
     * firstNode and secondNode is that of an Acronym of some sort.  If
         * acronym is not found in Abbreviation Table, method begins to create
         * its own acronyms from the Node's name, by searching for underscores
         * or capital letters.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @return an integer that denotes if one node is a concatenation of
     * the other node.
     * @see org.mitre.XMap.Equals
     * @since 1.0
     */

    public double acronymEquivalence(XMapNode firstNode,
                                                                                XMapNode secondNode) {
                String[] myAcronyms = new String[5];
                myAcronyms[1] = firstNode.getName();
                myAcronyms[2] = secondNode.getName();
                String firstName=myAcronyms[1];
                String secondName=myAcronyms[2];
                int firstLength=myAcronyms[1].length();
                int secondLength=myAcronyms[2].length();
                int i=0;
                char[] firstArray=myAcronyms[1].toCharArray();
                char[] secondArray=myAcronyms[2].toCharArray();
                String temp="";

                while((firstArray[i]<='Z') && (firstArray[i]>='A') && (i+1<firstLength)) {
                        i++;
                }
                if (i+1==firstLength) {
                        myAcronyms[1]=myAcronyms[1].toLowerCase();
                }
                i=0;
                while((secondArray[i]<='Z') && (secondArray[i]>='A') && (i+1<secondLength)) {
                        i++;
                }
                if (i+1==secondLength)
                        myAcronyms[2]=myAcronyms[2].toLowerCase();
                // The following for loops searches for capital
                // letters and tries to create acronyms with these

                for (i=0; i<firstLength; i++) {
                        if ((firstArray[i]<='Z') && (firstArray[i]>='A'))
                                temp=temp + firstArray[i];
                }
                myAcronyms[1]=temp;
                temp="";
                for (i=0; i<secondLength; i++) {
                        if ((secondArray[i]<='Z') && (secondArray[i]>='A'))
                                temp=temp + secondArray[i];
```

```
            }
            myAcronyms[2]=temp;
            // The following while loops searches for the character '_'
            // and tries to create acronyms.
            int stringIndex=0, counter=0;
            temp = String.valueOf(firstName.charAt(0));
            while (stringIndex!=-1) {
                    stringIndex=firstName.indexOf("_", counter);
                    if (stringIndex!=-1)
                            temp = temp+firstName.charAt(stringIndex+1);
                    counter = stringIndex+1;
            }
            myAcronyms[3]=temp;
            temp = String.valueOf(secondName.charAt(0));
            stringIndex=0;
            counter=0;
            while (stringIndex!=-1) {
                    stringIndex=secondName.indexOf("_", counter);
                    if (stringIndex!=-1)
                            temp = temp+secondName.charAt(stringIndex+1);
                    counter = stringIndex+1;
            }
            myAcronyms[4]=temp;
            // All acronyms are stored in myAcronyms.  The following
            // if statements get rid of single or empty strings.
            for(i=1;i<5;i++)
                    if (myAcronyms[i].length()==1)
                            myAcronyms[i]="";
            XMapNode tempNode1 = new XMapNode(myAcronyms[1], "");
            XMapNode tempNode2 = new XMapNode(myAcronyms[2], "");
            XMapNode tempNode3 = new XMapNode(myAcronyms[3], "");
            XMapNode tempNode4 = new XMapNode(myAcronyms[4], "");
            int value = abbreviationsEquivalence(tempNode1, secondNode) +
                    abbreviationsEquivalence(tempNode1, tempNode2) +
                    abbreviationsEquivalence(tempNode1, tempNode4) +
                    abbreviationsEquivalence(tempNode2, firstNode) +
                    abbreviationsEquivalence(tempNode2, tempNode3) +
                    abbreviationsEquivalence(tempNode3, tempNode4) +
                    abbreviationsEquivalence(tempNode3, secondNode) +
                    abbreviationsEquivalence(tempNode4, firstNode);
            if (value==0) {
                    // If there are no acronyms, we will manipulate the
                    // string and find possible words that match.
                    double myWeight;
                    myWeight=(stringManipulation(firstName, secondName))*WEIGHT;
                    return myWeight;
            }
            else
                    return EQ_TYPE_ACRONYM;
    }

    /**
     * Returns a weight which represents how closely related two
         * strings are to each other.  Method breaks down both names
         * smaller words, and proceeds to compare their respective
         * strings to each other.
     *
     * @param firstString String value of one domain to compare
     * @param secondString String value of another domain to compare
     * @return weight which represents how related two strings are.
     *
     * @see org.mitre.XMap.Equivalence
     * @since 1.0
     */

    public double stringManipulation(String firstString, String secondString) {
            String[] words1 = new String[11];
            String[] words2 = new String[11];
            words1[0]=firstString;
            words2[0]=secondString;
            int i=0;
            String temp="";
            int counter=1;
            char myChar='a';
```

```
		int myLength=words1[0].length();
		while (myLength!=i) {
			while ((myChar!='_') && (myLength>i) && ((myChar<'A') || (myChar>'Z'))) {
				temp=temp + String.valueOf(words1[0].charAt(i++));
				if (myLength!=i)
					myChar=words1[0].charAt(i);
			}
			if (temp!=words1[0]) {
				words1[counter]=temp;
				counter++;
				temp="";
			}
			if (myChar!='_')
				temp=String.valueOf(myChar);
			i++;
			if (myLength>i)
				myChar='a';

			else
				i=myLength;
		}
		i=0; temp=""; counter=1; myChar='a';
		myLength=words2[0].length();
		while (myLength!=i) {
			while ((myChar!='_') && (myLength>i) && ((myChar<'A') || (myChar>'Z'))) {
				temp=temp + String.valueOf(words2[0].charAt(i++));
				if (myLength!=i)
					myChar=words2[0].charAt(i);
			}
			if (temp!=words2[0]) {
				words2[counter]=temp;
				counter++;
				temp="";
			}
			if (myChar!='_')
				temp=String.valueOf(myChar);
			i++;
			if (myLength>i)
				myChar='a';
			else
				i=myLength;
		}
		i=1;
		int lettersMatched=0;
		int j=1;
		while (words1[i]!=null) {
			j=1;
			while (words2[j]!=null) {
				if (words1[i].toUpperCase()==words2[j].toUpperCase())
					lettersMatched=lettersMatched+words1[i].length();
				else {
					XMapNode tempNode1 = new XMapNode(words1[i].toUpperCase()
, "");
					XMapNode tempNode2 = new XMapNode(words2[j].toUpperCase()
, "");
					if (concactEquivalence(tempNode1, tempNode2)==EQ_TYPE_CON
CAT) {
						if (words1[i].length()>words2[j].length())
							lettersMatched=lettersMatched+words2[j].l
ength();
						else
							lettersMatched=lettersMatched+words1[i].l
ength();
					}
				}
				j++;
			}
			i++;
		}
		if (lettersMatched<4)
			lettersMatched=0;
		double myWeight=0;
		myWeight=((double)lettersMatched/(double)words1[0].length())+((double)lettersMatc
hed/(double)words2[0].length());
```

```
                // If program reaches this point and myWeight=0, then we see how well
                // the string matches up.
                if (myWeight==0) {
                        lettersMatched=stringMatch(words1[0], words2[0]);
                        myWeight=((double)lettersMatched/(double)words1[0].length())+((double)let
tersMatched/(double)words2[0].length());
                }
                return myWeight;
    }


    /**
     * Returns a number which represents the highest number of consecutive
     * characters that match in the string
     *
     * @param firstString String value of one domain to compare
     * @param secondString String value of another domain to compare
     * @return the highest number of consecutive characters that match
     *
     * @see org.mitre.XMap.Equivalence
     * @since 1.0
     */

    public int stringMatch(String firstString, String secondString) {
                int toffset=0;
                int i=0;
                int j=1;
                int lettersMatched=1;
                while (toffset<firstString.length()) {
                        while ((!firstString.regionMatches(true, toffset, secondString, i, j)) &&
 (i<secondString.length())) {
                                i++;
                        }
                        while (firstString.regionMatches(true, toffset, secondString, i, j)) {
                                j++;
                        }
                        if (lettersMatched<j-1)
                                lettersMatched=j-1;
                        toffset++;
                        i=0; j=1;
                }
                if (lettersMatched<4)
                        return 0;
                else
                        return lettersMatched;
    }


    /**
     * Returns an integer that denotes if the type of equivalence between
     * firstNode and secondNode is that of a concatenation.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @return an integer that denotes if one node is a concatenation of
     * the other node.
     * @see org.mitre.XMap.Equals
     * @since 1.0
     */

    public int concactEquivalence(XMapNode firstNode,
                                                                XMapNode secondNode) {
                String firstString = firstNode.getName();
                String secondString = secondNode.getName();
                if (firstString.indexOf(secondString) > -1 ||
                        secondString.indexOf(firstString) > -1) {
                        subWeight=64;
                        return EQ_TYPE_CONCAT;
                }
                else return 0;
    }
    /**
     * Returns an integer that denotes if the type of equivalence between
     * firstNode and secondNode.  Precision is measured by type and also
     * by the degree, which is specified in RATIO.  Note that Precision
         * Equals is not being used in this version.  Data is located in an
         * arrayList, and needs to be analyzed for this method to work.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @return an integer that denotes if one node is a concactenation of
     * the other node.
     * @see org.mitre.XMap.Equivalence, org.mitre.Xmap.LooseEquivalence.RATIO
     * @since 1.0
     */
    public int precisionEquivalence(Node firstNode,
                                    Node secondNode)  {

                //Given two nodes, check to see if they are both some type of
                //number.  If so, compare their types, ratio, and see how far
                //off they are by degree.
        try {
            double node1=Double.parseDouble(firstNode.getName());
            double node2=Double.parseDouble(secondNode.getName());
            if (node1==0) {
                if (node2<=10 && node2>=-10)
                    return EQ_TYPE_PRECISION;
                else
                    return 0;
            }
            if (node2==0) {
                if (node1<=10 && node1>=-10)
                    return EQ_TYPE_PRECISION;
                else
                    return 0;
            }
            double typeRatio = node1/node2;
            if (typeRatio<=1) {
                typeRatio = 1/typeRatio;
            }
            if ((typeRatio<=RATIO) && (typeRatio>=0)) {
                return EQ_TYPE_PRECISION;
            }
        }
        catch (NumberFormatException e)
            {}
        return 0;
    }

    /**
     * Returns an integer that denotes if the type of equivalence between
     * firstNode and secondNode is of Measurement.  Method searches within
     * two levels of the TaggedDiGraph and compares node to other values.
     * Thereafter, it uses MeasurementsTable to see if certain measurements
     * are related.  Note that measurementEquivalence is not being used currently
         * in this version.  Data analysis is necessary, and that functionality
         * hasn't been added yet.
     *
     * @param firstNode node from one domain to compare
     * @param secondNode node from another domain to compare
     * @return an integer that denotes if they are of the same type of
     * measurement.
     * @see org.mitre.XMap.Equals
     * @since 1.0
     */
    public int measurementEquivalence(XMapNode firstNode,
                                                                          XMapNode secondNode,
                                                                          TaggedDiGraph relations
Graph)
                throws NoNodeException {

                ArrayList arrayofSuccessorsa1=relationsGraph.successors(firstNode);
                ArrayList arrayofSuccessorsb1=relationsGraph.successors(secondNode);

                //After the while loops, ListofSuccessorsa2 & b2 will contain all
```

```
		//the nodes from the second layer of both domains.
		ArrayList arrayofSuccessorsa2=new ArrayList();
		ArrayList arrayofSuccessorsb2=new ArrayList();
		XMapNode tempNode;
		Iterator listofSuccessorsa1=arrayofSuccessorsa1.iterator();
		Iterator listofSuccessorsb1=arrayofSuccessorsb1.iterator();
		Iterator tmp=null;
		while (listofSuccessorsa1.hasNext()) {
			tempNode = (XMapNode) listofSuccessorsa1.next();
			tmp = relationsGraph.successors(tempNode).iterator();
			if (tmp!=null) {
				while (tmp.hasNext()) {
					arrayofSuccessorsa2.add(tmp.next());
				}
			}
		}
		while (listofSuccessorsb1.hasNext()) {
			tempNode = (XMapNode) listofSuccessorsb1.next();
			tmp = relationsGraph.successors(tempNode).iterator();
			if (tmp!=null) {
				while (tmp.hasNext()) {
					arrayofSuccessorsb2.add(tmp.next());
				}
			}
		}

		//4 comparisons must be done : a1->b1, a1->b2, b1->a2, a2->b2
		int temp=measurementTable.measurementEquivalence(arrayofSuccessorsa1, arrayofSucc
essorsb1, abbreviationTable);
		if (temp==1)
			return EQ_TYPE_UNITS;
		else {
			temp=measurementTable.measurementEquivalence(arrayofSuccessorsa1, arrayof
Successorsb2, abbreviationTable);
			if (temp==1)
				return EQ_TYPE_UNITS;
			else {
				temp=measurementTable.measurementEquivalence(arrayofSuccessorsa2,
 arrayofSuccessorsb1, abbreviationTable);
				if (temp==1)
					return EQ_TYPE_UNITS;
				else {
					temp=measurementTable.measurementEquivalence(arrayofSucce
ssorsa2, arrayofSuccessorsb2, abbreviationTable);
					if (temp==1)
						return EQ_TYPE_UNITS;
				}
			}
		}
		return 0;
	}
}
```

```
/*
 * ExactEquals.java
 *
 * Revision History
 *   initial version - 06/27/2000: Eddie Byon
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;
import org.mitre.XMap.AbstractEquals;
import org.mitre.XMap.NoNodeException;

/**
 * This class implements the ''exact'' notion of equals useful for prototyping
 * fuselets.
 * @author Eddie Byon
 * @version 1.0
 * @see org.mitre.XMap.StructuralAnalysis1
 */
public class ExactEquals extends AbstractEquals {
        /**
         * Constructs the default notion of ''loose'' equals.
         *
         * @since 1.0
         */

        public ExactEquals() {
                super();
                setDescription("Exact Equals");
        }

        /**
         * Returns an EQ_TYPE_EXACT if firstNode and secondNode given
         * are exactly equal.
         *
         * @param firstNode node from one domain to compare
         * @param secondNode node from another domain to compare
         * @param relationsGraph directed graph in which the nodes are compared
         * @return an integer that denotes the type of equivalence reached
         * @see org.mitre.XMap.Equals
         * @since 1.0
         */

        public int sEquals(Node firstNode,
                                          Node secondNode,
                                          TaggedDiGraph relationsGraph)
        throws NoNodeException {
                if (firstNode.equals(secondNode))
                        return EQ_TYPE_EXACT;
                else
                        return ;
        }
}
```

```
/*
 * HierarchyEdge.java
 *
 * Revision History
 *   initial version - 9/25/2000: Ashish Mishra
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Edge;
import org.mitre.XMap.XMapNode;

/**
 * A HierarchyEdge is a subclass of Edge which implements the XML
 * hierarchy structure.
 *
 * @author Ashish Mishra
 * @see org.mitre.XMap.XMapTDG
 */
public class HierarchyEdge extends Edge {

    /**
     * Create an Edge from newFromNode to newToNode in the same domain
     *
     * @param newFromNode the originating node of this Edge
     * @param newToNode the destinating node of this Edge
     * @param newEdgeTag the new tag value of this Edge
     * @since 1.0
     */
    public HierarchyEdge(XMapNode newFromNode, XMapNode newToNode)
        throws DomainsException{

        super(newFromNode, newToNode);
        if (!newFromNode.sameDomain(newToNode)) {
            throw new DomainsException("Nodes of different domains");
        }
    }
}
```

```
/**
 * Provides interface to read in XML linkbase files, converting Arcs
 * into AssociationEdges in the TDGraph.
 *
 * Revision history
 *   initial version - 09/25/2000: Ashish Mishra
 */

package org.mitre.XMap;

import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.XMapNode;
import org.mitre.XMap.DuplicateNodeException;
import org.mitre.XMap.NoNodeException;
import org.mitre.XMap.DuplicateEdgeException;
import org.mitre.XMap.LinkBase;

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.ArrayList;

import java.io.FileNotFoundException;
import java.io.Reader;
import java.io.FileReader;

import org.mitre.whiteboard.WhiteBag;
import org.mitre.simpleEx.XmlToBag;
import org.mitre.blackboard.AccessViolation;

public class Linkbase2DiGraph {
    //Hashtable uses bag's Id or bag Id concatinated with attribute name
    //  as key and holds the node as the Value
    //    protected Hashtable idNodes;

    //Each type of element has only one node it should use, this
    //  hashtable uses the "fullName" of the bag, see fullNameOfNode below,
    //  as the key and a single node to represent all like Elements
    //  as the value
    //    protected Hashtable allNodes;

    /** Table of association types, vs mapping numbers
     */
    private String[] assocTypes = {"equals",
                                   "lookup ",
                                   "synonym/acronym/abbreviation ",
                                   "conversion ",
                                   "reordering ",
                                   "abstraction ",
                                   "aggregation/decomposition ",
                                   "is-a ",
                                   "is-like ",
                                   "defaults "};

    public Linkbase2DiGraph() {
    }

    /*************************************************************
     *  Adds to a XMapTDG from a Linkbase File
     *
     *  @param fileName XML File to create Edges from
     *  @param tdg XMapTDG to append to
     *  @param domainName deprecated, domain is read in from attribute name
     *  @return XMapTDG representation of the XML file
     *  @exception FileNotFoundException
     *************************************************************/
    public void linkbase2Graph(String fileName, XMapTDG tdg, String domainName)
        throws FileNotFoundException {
        Hashtable rootLevel = loadXMLData(fileName);
        bags2Graph(rootLevel, tdg, domainName);
    }

    /*****************************************************
     * Changes bags to taggedDiGraph.
     * @param rootLevelBags HashTable where values are
     *   all the bags created from an XML file.
     * @return XMapTDG representing the
     *   collection of bags
     ****************************************************/
    private XMapTDG bags2Graph(Hashtable rootLevelBags, XMapTDG tdg, String domainName) {
        //Put every bag in the table if bags
        Enumeration e = rootLevelBags.keys();
        Hashtable idNodes = new Hashtable();
        while(e.hasMoreElements()) {
            WhiteBag bag = (WhiteBag)rootLevelBags.get(e.nextElement());
            //Only
            if(!bag.hasType("input-mapping-dtd")) {
                bagsToTable(bag, domainName, idNodes, tdg);
            }
        }
        tableToTDG(tdg, idNodes, domainName);
        return tdg;
    }


    /************************************************
     * Turns a bag and its attributes into a XMapNode and
     *   puts it into the idNodes hashtable.
     * @param rootBag Bag to add to the table.
     ************************************************/
    private void bagsToTable(WhiteBag rootBag, String domainName, Hashtable idNodes,
                             XMapTDG tdg) {
        XMapNode node;
        String id = rootBag.getId();
        String type = rootBag.getTypes()[0];

        if (type.equals("MappingArc")) {
            String fromName = null;
            String toName = null;
            String fromDomain = null;
            String toDomain = null;
            String[] assocType = null;
            String myAssocType=null;
            Enumeration attrNames;
            String attr, nodeString;
            Object[] attrValues = null;
            XMapNode fromNode, toNode;
            int i, j;

            try {
                attrNames = rootBag.getNamesAndRoles();

                //  go through all attributes of the bag.
                //  these attributes can be references to other bags,
                //  or actual attributes
                while(attrNames.hasMoreElements()) {
                    attr = (String)attrNames.nextElement();
                    if (attr.equalsIgnoreCase("xlink:from") ||
                        attr.equalsIgnoreCase("xlink:to")) {
                        attrValues = rootBag.getValues(attr);
                    //          numValues = attrValues.length;
                        nodeString = (String) attrValues[0];
                        i = nodeString.indexOf(":");
                        j = nodeString.lastIndexOf(":");

                        if (attr.equalsIgnoreCase("xlink:from")) {
                            //                      fromDomain = nodeString.substring(0, i) + ".x
ml";
                            fromDomain = nodeString.substring(0, i);
                            fromName = nodeString.substring(i+1, j);
                        } else {
                            //                      toDomain = nodeString.substring(0, i) + ".xml
";
                            toDomain = nodeString.substring(0, i);
                            toName = nodeString.substring(i+1, j);
                        }
                    }
                    if (attr.equalsIgnoreCase("genericType")) {
                        attrValues = rootBag.getValues(attr);
                    myAssocType = (String) attrValues[0];
                }
            }
            fromNode = tdg.getNode(fromName, fromDomain);
            if (fromNode == null) {
                //System.out.println("Creating this node: " + fromDomain +
                //      ":" + fromName);
                fromNode = new XMapNode(fromName, fromDomain);
                tdg.addNode(fromNode);
            }
            toNode = tdg.getNode(toName, toDomain);
            if (toNode == null) {
                //System.out.println("Creating this node: " + toDomain +
                            //       ":" + toName);
                toNode = new XMapNode(toName, toDomain);
                tdg.addNode(toNode);
            }
            if (fromNode.sameDomain(toNode)) {
                System.err.println("Error reading linkbase file");
                System.exit(1);
            }
            int assoc = 0;
//          // myAssocType contains all possible types so we must parse it now.
//          int count=0;
//          for(i=0;i<assocTypes.length;i++) {
//                  if (assocTypes[i].regionMatches(true, 0, myAssocType, 0, myAssocType.leng
th())) {
//                          assocType[count++]=assocTypes[i];
//                          System.out.println(assocTypes[i]);
//                  }
//          }
//          for(i=0;i<count;i++) {
//                  assoc+=associationInteger(assocType[i]);
//          }
            assoc = LinkBase.getNumericType(myAssocType.trim());

            //System.out.println("READING LB");
            //System.out.println(assoc + " " + assocType);
            AssociationEdge newAssoc = new AssociationEdge(fromNode, toNode, assoc);
            if (tdg.hasEdge(fromNode, toNode)) {
                // What to do here?  New edge conflicts with pre-existing association
                tdg.removeEdge(fromNode, toNode);  //let's be brutal
            }

            tdg.addEdge(fromNode, toNode, newAssoc);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}


/*********************************************
 * Takes all Nodes from the table of nodes
 *   and combines them to create the TDG.
 * @return XMapTDG representing the XML
 *   Document where each like element shares a node.
 *********************************************/
protected void tableToTDG(XMapTDG tdg, Hashtable idNodes, String domainName) {
    Hashtable allNodes = new Hashtable();
    Enumeration ids = idNodes.keys();
    XMapNode oldNode, newNode;
    int j=0;
    try {
        while(ids.hasMoreElements()) {
            oldNode = (XMapNode)idNodes.get(ids.nextElement());
            newNode = getNode(oldNode, tdg, allNodes, domainName);

            ArrayList data = oldNode.getData();
            int dataLength = data.size();
            for(int i = 0; i < dataLength; i++) {
                newNode.addData(data.get(i));
            }
            }
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }


    /***********************************************
     * Since more than one node should be the same, this method
     *   takes a node as a parameter and returns the node that it
     *   shares with other like nodes.&nbsp;Also adds Node to the
     *   DiGraph.&nbsp; Use this method whenever you want to
     *   manipulate the contents of a general node.
     ***********************************************/
    protected XMapNode getNode(XMapNode thisNode, XMapTDG tdg, Hashtable allNodes, String domainN
ame) throws
        DuplicateNodeException, NoNodeException, DuplicateEdgeException  {
        if(thisNode == null)
            return null;
        String fullName = fullNameOfNode(thisNode);
        if(allNodes.containsKey(fullName))
            return (XMapNode)allNodes.get(fullName);
        else {
            XMapNode parent = getNode(thisNode.getParent(), tdg, allNodes, domainName);
            XMapNode newNode = new XMapNode(thisNode.getName(), parent,
                                            domainName);
            allNodes.put(fullName, newNode);
            return newNode;
        }
    }

    /*********************************************
     * Concatinates the name of a node with its
     *   parent if it exists.
     * @param node Node that you want the full name of.
     * @return String representing the fullname of the Node
     *********************************************/
    protected String fullNameOfNode(XMapNode node) {
        if(node == null)
            return "";
        if(node.getParent() == null)
            return node.getName();
        else
            return node.getName() + node.getParent().getName();
    }

    /************************************************************
     * loads XML data from a file, puts all Nodes into
     *    a hashtable.
     *
     * @param fileName Name of XML file to load.
     * @return Hashtable where the values are all the bags.
     ************************************************************/
    private Hashtable loadXMLData(String fileName)
        throws FileNotFoundException {

        Hashtable hash = new Hashtable();
        Reader reader = new FileReader(fileName);
        XmlToBag parser = new XmlToBag();

        try {
            parser.parse(reader, hash);
        }
        catch (AccessViolation av) {
            av.printStackTrace();
        }

        return hash;
    }

    private int associationInteger(String assocType) {
        int i, mapping = 0;
        for(i=0; i<assocTypes.length;i++) {
            if (assocType.equals(assocTypes[i])) {
                        mapping+=Math.pow(2, i);
            }
        }
        if (mapping>0) {
            return mapping;
        } else {
            // Unrecognized association?
            return Equals.EQ_TYPE_NORMAL;
        }
    }
}
```

```
/*
 * LookupTable.java
 *
 * Revision History
 *   initial version - 06/21/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

/**
 * This interface abstracts away the notion of ''looking'' up something in
 * a non-mutating way.  This allows methods to universally look up values, be
 * it from a table, database, or chart of different types.  Implementations of
 * this interface act as the ''glue'' between that information and XMap.
 *
 * @author David Wang
 * @version 1.0
 */
public interface LookupTable {
        /**
         * Returns the value to which this key maps to.  The value can be multiple
         * objects.
         * <P>
         * The key is required to not be null.
         *
         * @param key key whose associated value is to be returned
         * @return the value to which this key maps to
         * @exception NotFoundException if key does not exist in the map
         * @since 1.0
         */
        public Object lookFor(Object key)
                throws NotFoundException;
}
```

```java
/*
 * DTDtoDiGraph.java
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.Iterator;
//import java.util.ArrayList;
import java.util.NoSuchElementException;


/**
 * This class provides methods to generate random data based on
 * parameters specified at the tags
 *
 * @author Ashish Mishra
 */


public class MakeData {

    public static int NUM_DATA_VALUES=100;

    public MakeData() {
    }

    public static void makeData(XMapTDG tdg) {

        Iterator allNodes = tdg.nodes();
        XMapNode node;
        String nodename;
        int par1, par2;
        String outval;
        double chartype;

        while (allNodes.hasNext()) {
            try {
                node = (XMapNode) allNodes.next();
                nodename = node.getName();

                if (nodename.startsWith("MakeData")) {
                    for (int i = 0; i<NUM_DATA_VALUES; i++) {
                        if (nodename.startsWith("MakeDataNumber")) {
                            // Number of digits from 1st param
                            par1 = (new Integer(nodename.substring(14, 15)))
                                .intValue();
                            // Number of decimal places from 2nd param
                            par2 = (new Integer(nodename.substring(15, 16)))
                                .intValue();

                            outval = String.valueOf
                                (Math.rint(Math.random() *
                                           Math.pow(10, par1)) /
                                 Math.pow(10, par2));

                        } else {
                            // Number of characters from 1st param
                            par1 = (new Integer(nodename.substring(14)))
                                .intValue();

                            outval = "";
                            for (int j = 0; j<par1; j++) {
                                chartype = Math.random();
                                if (chartype<.05) {  // space
                                    outval += " ";
                                } else if (chartype<.15) { // capital letter
                                    outval += (char)('A' +
                                                (char)Math.floor
                                                (Math.random() * 26));
                                } else {
                                    outval += (char)('a' +
                                                (char)Math.floor
                                                (Math.random() * 26));
                                }
                            }
                        }
                        node.getParent().addData(outval);
                    }
                    tdg.removeEdge(node.getParent(), node);
                }
            } catch (NoSuchElementException e) {
            } catch (IndexOutOfBoundsException e) {
                e.printStackTrace();
            } catch (NoEdgeException e) {
                //e.printStackTrace();
                // why does this get thrown?
            }
        }
    }

}
```

```
/*
 * Node.java
 *
 * Revision History
 *       initial version - 06/08/2000: David Wang
 *   changed version - 07/08/2000: Eddie Byon
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.NodeTag;

/**
 * A Node is a data abstraction object that has a name and other associated
information (its tag).
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 */
public class Node {
    /** The name of this Node. */
    private String name;

    /** The tag associated with this Node. */
    private NodeTag tag;

    /**
     * Creates a Node with newName as its name and empty string for its tag value.
     *
     * @param newOwner the new name of this Node
     * @since 1.0
     */
    public Node(String newName)  {
        this(newName, "");
    }

    /**
     * Creates a Node with newName as its name and newTag as
     * the data in its NodeTag
     *
     * @param newName the new owner of this Node
     * @param newTag the new tag value of this Node
     * @since 1.0
     */
    public Node(String newName, String newNodeTagData) {
        setName(newName);
        setTag(new NodeTag(this, newNodeTagData));
    }

    /**
    * Creates a Node with newName as its name and newTag as its tag value.
    *
    * @param newName the new owner of this Node
    * @param newTag the new tag value of this Node
    * @since 1.0
    */
    public Node(String newName, NodeTag newTag) {
        setName(newName);
        setTag(newTag);
    }

    /**
     * Returns the name of this Node.
     *
     * @return the name of this Node
     * @since 1.0
     */
    public String getName() {
        return name;
    }

    /**
     * Replaces the value of this Node's name with newName.
     *
     * @param newName the new name of this Node
     * @since 1.0
     */
    public void setName(String newName) {
        name = newName;
    }

    /**
     * Returns the tag value of this Node.
     *
     * @return the tag value of this Node
     * @since 1.0
     */
    public NodeTag getTag() {
        return tag;
    }

    /**
     * Replaces the value of this Node's tag with newTag.
     *
     * @param newTag the new tag of this Node
     * @since 1.0
     */
    public void setTag(NodeTag newTag) {
        tag = newTag;
    }
}
```

```
/*
 * NodeTag.java
 *
 * Revision History
 *       initial version - 06/07/2000: David Wang
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import org.mitre.XMap.Tag;

/**
 * A NodeTag is an type of Tag whose value is constrained to be a String.
 *
 * @author David Wang
 * @version 1.0
 * @see org.mitre.XMap.TaggedDiGraph
 */
public class NodeTag extends Tag {
        /**
         * Use NodeTag(Object, String) instead. It is not desirable to instantiate
         * a NodeTag without an owner and tag data.
         *
         * @exception Error can't use this constructor
         * @see #NodeTag(Object, String)
         */
        protected NodeTag() {
                super();
        }

        /**
         * Creates a NodeTag with newNodeOwner as its owner and newNodeTagData as
         * its tag value string.
         *
         * @param newOwner the new owner of this NodeTag
         * @param newNodeTagData the new tag value string of this NodeTag
         * @since 1.0
         */
        public NodeTag(Object newNodeOwner, String newNodeTagData) {
                super(newNodeOwner, newNodeTagData);
                //setOwner(newNodeOwner);
                //setTagData(newNodeTagData);
        }

        /**
         * Replaces the value of this NodeTag with newNodeTagData.
         *
         * @param newNodeTagData the new tag value string of this NodeTag
         * @since 1.0
         */
        public void setTagString(String newNodeTagData) {
                setTagData(newNodeTagData);
        }

        /**
         * Returns the value of this NodeTag.
         *
         * @return the tag value string of this NodeTag
         * @since 1.0
         */
        public String getTagString() {
                return (String)getTagData();
        }
}
```

```
/*
 * StructuralAnalysis2.java
 *
 * Revision History
 *   initial version - 06/19/2000: David Wang
 *           revised - 9/25/2000: Ashish Mishra
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */


package org.mitre.XMap;

import java.util.Iterator;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.NoSuchElementException;

import org.mitre.XMap.AbstractAnalysisAlgorithm;
import org.mitre.XMap.EquivalenceAnalysis;
import org.mitre.XMap.UniqueHashMap;
import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.AssociationEdge;
import org.mitre.XMap.HierarchyEdge;
import org.mitre.XMap.NoEdgeException;
import org.mitre.XMap.DuplicateEdgeException;


/**
 * An implementation of an analysis algorithm that performs association analysis
 * based on structural (i.e.&nbsp; graph theory) characteristics.  The
 * characteristics of interest include:
 * <UL>
 *   <LI>Cycles in the directed graph</LI>
 *   <LI>''Hierarchies'' flattened into a directed graph</LI>
 *   <LI>Conflicting associations</LI>
 *   <LI>Proximity of nodes to known associations, especially if more
 * than one nearby association exists
 * </UL>
 *
 * @author Ashish Mishra
 * @see org.mitre.XMap.AbstractAnalysisAlgorithm
 * @see org.mitre.XMap.XMapTDG
 */
public class SA2 extends AbstractAnalysisAlgorithm
{
    /**
     * Constructs a new instance of this structural analysis algorithm.  This
     * algorithm implements the XMap strategies as outlined in David Wang's
     * June 2000 thesis.
     */

//      /* Number levels up & down to rescore
//       */
//      private int checkLevels;
//      /* Amount score drops off with increasing distance from known associations
//       */
//      private double scoreDropOff;
//      /* Threshold above which to discard associations as insignificant
//       */
//      private double scoreThreshold;
//      /* Rootnodes for the domains being compared
//       */
    private XMapNode firstRoot;
    private XMapNode secondRoot;
    public static int MAX_POWS = 8;
    public static double scoreThreshold = 0.7;
    // first level modification to scores
    public static double scoreChange = 0.69;
    // Drop off in modification as you go farther away
    public static double scoreDropOff = 0.8;

    /* Creates a new instance of this algorithm, using default parameters
     * of 2, 0.65, 0.7, and operating on the domains containing root1 & root2
     * respectively.  Emphatically, the given defaults may not work well
     * with all applications, and actual values must be carefully chosen.
     *
     * @param root1 Root node for one of the comparison domains
     * @param root2 Root node for the other domain
     */
    public SA2(XMapNode root1, XMapNode root2)
    {
//      this(2, 0.65, 0.7, root1, root2);
//      // What are good defaults for these parameters?
//     }

//     /* Creates a new instance of this algorithm, using the given parameters,
//      * and operating on the domains containing root1 & root2
//      * respectively.  The parameters are as follows:
//      * @param cl Number levels up & down to rescore.  Increasing
//      * this will increase number of edges thrown up.
//      * @param sdo Amount score drops off with increasing distance
//      * from known associations.  Increasing this will increase number of
//      * edges thrown up.
//      * @param st Threshold above which edges are regarded as potential
//      * asociations.  Increasing this will (obviously) reduce the number
//      * of edges thrown up.
//      * @param root1 Root node for one of the comparison domains
//      * @param root2 Root node for the other domain
//      */
//     public StructuralAnalysis1(int cl, double sdo, double st,
//                                XMapNode root1, XMapNode root2)
//     {
        super();
        setAlgorithmName ("Structural Analysis 2");
        setDefaultEquivalenceDefn(new EquivalenceAnalysis());

//      checkLevels = cl;
//      scoreDropOff = sdo;
//      scoreThreshold = st;
        firstRoot = root1;
        secondRoot = root2;
    }


    public Iterator analyze(XMapTDG graph)
    {
        // load nodes into two vectors.
        ArrayList dom1nodes = new ArrayList();
        ArrayList dom2nodes = new ArrayList();
        Iterator nodes = graph.nodes();
        XMapNode node = null;
        ArrayList discoveredEdges = new ArrayList();

        while (nodes.hasNext()) {
            try {
                node = (XMapNode) nodes.next();
            } catch (NoSuchElementException e) {
                // can't happen
            }
            if (node.sameDomain(firstRoot)) {
                dom1nodes.add(node);
            } else if (node.sameDomain(secondRoot)) {
                dom2nodes.add(node);
            }
        }
        int[][] adjMat1 = makeAdjMatrix(dom1nodes, graph);
        int[][] adjMat2 = makeAdjMatrix(dom2nodes, graph);

        int[] relArray = popRelArray(dom1nodes, dom2nodes, graph);

        int[][][] expmat1 = getMatrixExponents(adjMat1);
        int[][][] expmat2 = getMatrixExponents(adjMat2);
    // the meat of the routine
    compareNumPaths(expmat1, expmat2, graph, relArray,
                    dom1nodes, dom2nodes);

    retainGoodAssociations(graph);


    // this actually doesn't return anything useful, all the important
    // information is already encapsulated in the graph
    return discoveredEdges.iterator();
}

private int[][][] getMatrixExponents(int[][] mat)
{
    int[][][] em1 = new int[MAX_POWS][][];
    em1[1] = mat;
    for (int i=2; i<MAX_POWS; i++) {
        System.out.println(i);
        em1[i] = multiplyMatrices(em1[i-1], mat);
    }
    return em1;
}

private int[] popRelArray(ArrayList dom1nodes, ArrayList dom2nodes,
                          XMapTDG graph)
{
    //Populates the relationship array with currently known relationships
    XMapNode nodei, nodej;
    AssociationEdge ae;
    int[] relArray = new int[dom1nodes.size()];

    try {
        for (int i = 0; i<dom1nodes.size(); i++) {
            relArray[i]=-1;
            nodei = (XMapNode) dom1nodes.get(i);
            for (int j = 0; j<dom2nodes.size(); j++) {
                nodej = (XMapNode) dom2nodes.get(j);
                if (!nodei.isAttribute() &&
                    !nodej.isAttribute() &&
                    graph.hasAssociation(nodei, nodej)) {

                    ae = (AssociationEdge)
                        graph.getEdge(nodei, nodej);
                    if (ae.isConfidentMapping()) {
                        ae.printEdge();
                        relArray[i]=j;
                    }
                }
            }
        }
    } catch (NoSuchElementException e) { // shouldn't happen
    } catch (NoEdgeException e) { // shouldn't happen
    }
    return relArray;
}

    private int[][] makeAdjMatrix(ArrayList domNodes, XMapTDG graph)
{
    // adjMatrix[i][j] is 1 if an edge runs from node i to node j.
    // 0 otherwise
    XMapNode nodei, nodej;
    int n = domNodes.size();
    int[][] mat = new int[n][n];

    try {
        for (int i = 0; i<n; i++) {
            nodei = (XMapNode) domNodes.get(i);
            for (int j = 0; j<n; j++) {
                nodej = (XMapNode) domNodes.get(j);

                if (graph.hasEdge(nodei, nodej)) {
                    mat[i][j]=1;
                } else {
                    mat[i][j]=0;
                    }
                }
            }
        } catch (NoSuchElementException e) {
            // shouldn't happen
        }
        return mat;
    }

    private int[][] multiplyMatrices(int[][] mat1, int[][] mat2)
    {
        // assumes matrices are square and same size
        int n = mat1.length;
        int prod[][] = new int[n][n];

        for (int i = 0; i<n; i++) {
            for (int j = 0; j<n; j++) {
                prod[i][j] = 0;
                for (int k = 0; k<n; k++) {
                    prod[i][j] += mat1[i][k]*mat2[k][j];
                }
                System.out.print(prod[i][j] + " ");
            }
            System.out.println();
        }
        return prod;
    }

    private void compareNumPaths(int[][][] expmat1, int[][][] expmat2,
                                 XMapTDG graph, int[] relArray,
                                 ArrayList d1, ArrayList d2) {
        int n1, n2;
        double schange = scoreChange/scoreDropOff;
        int[][] emat1, emat2;

        for (int i=1; i<MAX_POWS; i++) {
            schange *= scoreDropOff;
            //schange = java.lang.Math.pow(scoreInc, i);
            emat1 = expmat1[i];
            emat2 = expmat2[i];

            for (int j=0; j<relArray.length; j++) {
                if (relArray[j]>-1) {

                    for (int k = 0; k<emat1.length; k++) {
                        for (int l = 0; l<emat2.length; l++) {
                            n1 = emat1[j][k];
                            n2 = emat2[relArray[j]][l];

                            //                      System.out.println(i + " " + j + " " + k + "
" + l + " " + n1 + " " + n2);
                            if (n1 > 0 && n1 == n2) {
                                updateEdgeScore((XMapNode) d1.get(k),
                                                (XMapNode) d2.get(l),
                                                graph, schange);
                                updateEdgeScore((XMapNode) d2.get(l),
                                                (XMapNode) d1.get(k),
                                                graph, schange);
                                System.out.println(((XMapNode) d1.get(k)).
                                                   getName() + "\t" +
                                                   ((XMapNode) d2.get(l)).
                                                   getName() + "\n" +
                                                   ((XMapNode) d1.get(j)).
                                                   getName() + "\t" +
                                                   ((XMapNode)
                                                    d2.get(relArray[j])).
                                                   getName());
                            }
                            n1 = emat1[k][j];
                            n2 = emat2[l][relArray[j]];

                            if (n1 > 0 && n1 == n2) {
                                updateEdgeScore((XMapNode) d1.get(k),
                                                (XMapNode) d2.get(l),
                                            graph, schange);
                              updateEdgeScore((XMapNode) d2.get(l),
                                              (XMapNode) d1.get(k),
                                              graph, schange);
                          }
                      }
                  }
              }
          }
      }
      // pretty simple huh?
  }

  /* Updates the scores on edges, as per scoreChange
   */
  private void updateEdgeScore(XMapNode node1, XMapNode node2,
                               XMapTDG graph, double scoreChange) {
      AssociationEdge edge;

      System.out.println("Updating this edgeScore");
      System.out.println(node1.getName() + "->" + node2.getName() +
                         "\t" + scoreChange);
      try {
          if (graph.hasAssociation(node1, node2)) {
              edge = (AssociationEdge) graph.getEdge(node1, node2);
              // Scores accumulate.
              // Need the next line due to roundoff-type errors
              if (edge.getScore() < 1.0)
                  edge.setScore(combineScores(edge.getScore(), scoreChange));
          } else {
              graph.addEdge(node1, node2,
                            new AssociationEdge(node1, node2, 0,
                                                scoreChange));
          }
      } catch (DuplicateEdgeException e) {
          // Means there is already an edge representing e.g.
          // a heirarchy.  Don't need to do anything.
      } catch (NoEdgeException e) {
          e.printStackTrace();
          // Thrown by TaggedDiGraph.getEdge,
          //  -- Shouldn't happen
      } catch (NoNodeException e) {
          e.printStackTrace();
          // Thrown by TDG.addEdge  -- Shouldn't happen
      }
  }

  /* A function that combines scores that accrue from two distinct events
   * (such as nearby known associations)
   */
  private double combineScores(double d1, double d2) {
      // Discussed characteristics of this function with Mike.
      // Using below for the moment.

      return d1 + d2 - d1*d2;
  }

  private void retainGoodAssociations(XMapTDG graph) {
      AssociationEdge edge = null;
      Iterator edges;
      double thisScore;

      // now find which edges have score over threshold...
      // do this in XMap instead?
      edges = graph.associationEdges().iterator();
      while (edges.hasNext()) {
          try {
              edge = (AssociationEdge) edges.next();
              thisScore = edge.getScore();
              if (thisScore < scoreThreshold) {
                  graph.removeEdge(edge);
              }
          } catch (NoSuchElementException ex) {
              // Thrown by Iterator.next  -- Shouldn't happen
```

```
            } catch (NoEdgeException ex) {
               // Thrown by graph.removeEdge  -- Shouldn't happen
            }
         }
      }


}
```

```
/*
 * StructuralAnalysis1.java
 *
 * Revision History
 *   initial version - 06/19/2000: David Wang
 *           revised - 9/25/2000: Ashish Mishra
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */


package org.mitre.XMap;

import java.util.Iterator;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.NoSuchElementException;

import org.mitre.XMap.AbstractAnalysisAlgorithm;
import org.mitre.XMap.EquivalenceAnalysis;
import org.mitre.XMap.UniqueHashMap;
import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.AssociationEdge;
import org.mitre.XMap.HierarchyEdge;
import org.mitre.XMap.NoEdgeException;
import org.mitre.XMap.DuplicateEdgeException;


/**
 * An implementation of an analysis algorithm that performs association analysis
 * based on structural (i.e.&nbsp; graph theory) characteristics.  The
 * characteristics of interest include:
 * <UL>
 *   <LI>Cycles in the directed graph</LI>
 *   <LI>''Hierarchies'' flattened into a directed graph</LI>
 *   <LI>Conflicting associations</LI>
 *   <LI>Proximity of nodes to known associations, especially if more
 * than one nearby association exists
 * </UL>
 *
 * @author David Wang
 * @author Ashish Mishra
 * @see org.mitre.XMap.AbstractAnalysisAlgorithm
 * @see org.mitre.XMap.XMapTDG
 */
public class StructuralAnalysis1 extends AbstractAnalysisAlgorithm
{
    /**
     * Constructs a new instance of this structural analysis algorithm.  This
     * algorithm implements the XMap strategies as outlined in David Wang's
     * June 2000 thesis.
     */

    /* Number levels up & down to rescore
     */
    private int checkLevels;
    /* Amount score drops off with increasing distance from known associations
     */
    private double scoreDropOff;
    /* Threshold above which to discard associations as insignificant
     */
    private double scoreThreshold;
    /* Rootnodes for the domains being compared
     */
    private XMapNode firstRoot;
    private XMapNode secondRoot;

    /* Creates a new instance of this algorithm, using default parameters
     * of 2, 0.65, 0.7, and operating on the domains containing root1 & root2
     * respectively.  Emphatically, the given defaults may not work well
     * with all applications, and actual values must be carefully chosen.
     *
     * @param root1 Root node for one of the comparison domains
     * @param root2 Root node for the other domain
     */
    public StructuralAnalysis1(XMapNode root1, XMapNode root2)
    {
        this(2, 0.65, 0.7, root1, root2);
        // What are good defaults for these parameters?
    }

    /* Creates a new instance of this algorithm, using the given parameters,
     * and operating on the domains containing root1 & root2
     * respectively.  The parameters are as follows:
     * @param cl Number levels up & down to rescore.  Increasing
     * this will increase number of edges thrown up.
     * @param sdo Amount score drops off with increasing distance
     * from known associations.  Increasing this will increase number of
     * edges thrown up.
     * @param st Threshold above which edges are regarded as potential
     * asociations.  Increasing this will (obviously) reduce the number
     * of edges thrown up.
     * @param root1 Root node for one of the comparison domains
     * @param root2 Root node for the other domain
     */
    public StructuralAnalysis1(int cl, double sdo, double st,
                               XMapNode root1, XMapNode root2)
    {
        super();
        setAlgorithmName ("Structural Analysis 1");
        setDefaultEquivalenceDefn(new EquivalenceAnalysis());

        checkLevels = cl;
        scoreDropOff = sdo;
        scoreThreshold = st;
        firstRoot = root1;
        secondRoot = root2;
    }


    public Iterator analyze(XMapTDG graph)
    {
        /** Run equals analysis using known associations
         * as a starting point.
         *
         * For high-scoring edges, this also modifies the original graph
         * to add an edge giving the score.  Is this a good
         * model?  Probably.
         * On the other hand, maybe this method can be repeatedly iterated
         * and actually use the scores in further iterations.
         * Note:  this returns edges in both directions.  This shouldn't be
         * a problem, though.<P>
         * Actually performs the structural analysis, which basically tries to make
         * associations based on several strategies and notions.  They include:
         * <UL>
         *   <LI>Cycles in the directed graph, which indicate some sort of
         *       aggregation or generalization conflict due to similar sorts of
         *       information being organized differently in the corresponding data
         *       domains</LI>
         *   <LI>''Hierarchies'' flattened into a directed graph, whereby if two
         *       leaf nodes are associated, their parents/grandparents/etc are often
         *       relatable in some manner (or at least speculated)</LI>
         *   <LI>Conflicting associations are either ''A implies B and A does not
         *       implie B'' or ''A implies B but B does not imply A''.  The latter
         *       can be speculated as a possible association while the former is
         *       thornier and needs a body of knowledge to resolve</LI>
         * </UL>
         *
         * The resulting associations are returned as an Iterator.
         *
         * @param graph the directed graph to analyze
         * @return an Iterator of association Edges discovered
         * @since 1.0
         */
```

```java
        //BUGBUG: do analysis here, using the appropriate equals definition

        ArrayList discoveredEdges = new ArrayList();
        // Contains ArrayList of discovered Associations

        AssociationEdge edge = null;
        UniqueHashMap nodesBeginAssoc, nodesEndAssoc;
        XMapNode nodeBA, nodeEA, firstNode, secondNode, chainFromRoot;
        Iterator nodesBA, nodesEA, edges;
        Set nodesEASet;
        double scoreChange, thisScore;
        Iterator knownAssocs = graph.associationEdges().iterator();

        // Cycle through known associations
        while (knownAssocs.hasNext()) {
            try {
                edge = (AssociationEdge) knownAssocs.next();
            } catch (NoSuchElementException e) { // Shouldn't happen
            }
            // Only look at definite or semi-definite edges
            //  we may change this paradigm later!

            // This is where we combine the scores.
            //Figure out a good way to do this.
            edge.setScore(combineScores(edge.getEquivalenceScore(),
                                        edge.getDataScore()));

            if (edge.isConfidentMapping()) {

                firstNode = (XMapNode) edge.getFromNode();
                secondNode = (XMapNode) edge.getToNode();

                if (firstNode.sameDomain(firstRoot)) {
                    chainKnownAssociations(firstNode, secondNode, firstRoot, graph);
                }
                if (firstNode.sameDomain(secondRoot)) {
                    chainKnownAssociations(firstNode, secondNode, secondRoot, graph);
                }
                if ((firstNode.sameDomain(firstRoot) && secondNode.sameDomain(secondRoot)) ||
                    (firstNode.sameDomain(secondRoot) && secondNode.sameDomain(firstRoot))) {
                //              edge.printEdge();
                    nodesBeginAssoc = scoreNodes(firstNode, graph, secondNode);
                    nodesEndAssoc = scoreNodes(secondNode, graph, firstNode);

                    nodesBA = nodesBeginAssoc.keySet().iterator();
                    nodesEASet = nodesEndAssoc.keySet();

                    while (nodesBA.hasNext()) {
                        nodesEA = nodesEASet.iterator();
                        try {
                            nodeBA = (XMapNode) nodesBA.next();
                            while (nodesEA.hasNext()) {
                                nodeEA = (XMapNode) nodesEA.next();

                                /** Score of edge is product of scores of both nodes
                                 * (which fall away exponentially with distance)
                                 * seems a reasonable metric to use.
                                 */
                                scoreChange = (((Double) nodesBeginAssoc.get(nodeBA))
                                               .doubleValue() *
                                               ((Double) nodesEndAssoc.get(nodeEA))
                                               .doubleValue()) *
                                    0.99;

                                // edges are added in both directions.  Both edges are
                                // symmetrically modified.      (??)
                                updateEdgeScore(nodeBA, nodeEA, graph, scoreChange);
                                //     updateEdgeScore(nodeEA, nodeBA, graph, scoreChange);
                            }
                        } catch (NotFoundException ex) {
                                // Thrown by UniqueHashMap.get  -- Shouldn't happen
                        } catch (NoSuchElementException ex) {
                                // Thrown by Iterator.next  -- Shouldn't happen
                        }
                }
            }
        } // Matches if(edge.isConfident...
    } // Matches while(knownAssocs.hasNext...

    // now find which edges have score over threshold...
    // do this in XMap instead?
    edges = graph.associationEdges().iterator();
    while (edges.hasNext()) {
        try {
            edge = (AssociationEdge) edges.next();
            thisScore = edge.getScore();
            if (thisScore < scoreThreshold) {
                graph.removeEdge(edge);
            }
        } catch (NoSuchElementException ex) {
                        // Thrown by Iterator.next  -- Shouldn't happen
        } catch (NoEdgeException ex) {
                        // Thrown by graph.removeEdge  -- Shouldn't happen
        }
    }

    // this actually doesn't return anything useful, all the important
    // information is already encapsulated inthe graph
    return discoveredEdges.iterator();
}


/* Chains known associations across domains, so if A->B and B->C are
 * known associations, this adds A->C as an association
 */
private void chainKnownAssociations(XMapNode fromNode, XMapNode toNode,
                                    XMapNode chainFromRoot, XMapTDG graph) {
    Iterator nextAssociations;
    AssociationEdge tryEdge;
    XMapNode destNode;

    try {
        nextAssociations = graph.confidentMappings(toNode).iterator();
        while (nextAssociations.hasNext()) {
            destNode = (XMapNode) nextAssociations.next();
            if (!destNode.sameDomain(chainFromRoot)) {
                //Chain the next association
                if (!graph.hasEdge(fromNode, destNode)) {
                    graph.addEdge(fromNode, destNode,
                              new AssociationEdge(fromNode, destNode, 0, 0.99));
    System.out.println("Chaining " + fromNode.getName() + " " + destNode.getName());
                    chainKnownAssociations(fromNode, destNode,
                                        chainFromRoot, graph);
                } else {
                    // only change existing edge if it's an association
                    if (graph.hasAssociation(fromNode, destNode)) {
                        tryEdge = (AssociationEdge)
                            graph.getEdge(fromNode, destNode);
                        if (!tryEdge.isConfidentMapping()) {
                            // if it isn't already marked, mark as
                            // confident and chain from it
                            tryEdge.setScore(0.99);
    System.out.println("Chaining " + fromNode.getName() + " " + destNode.getName());
                            chainKnownAssociations(fromNode, destNode,
                                                chainFromRoot, graph);
                        }
                    }
                }
                // Must terminate if there are a finite # of domains
            }
        }
    } catch (NoSuchElementException e) {
        // Shouldn't happen
    } catch (NoNodeException e) {
        // thrown by graph.mappingSuccessors(), addEdge() --
        // Shouldn't happen
    } catch (DuplicateEdgeException e) {
        // thrown by graph.addEdge() -- Shouldn't happen
        } catch (NoEdgeException e) {
            // thrown by graph.getEdge() -- Shouldn't happen
        }
    }


    /* Updates the scores on edges, as per scoreChange
     */
    private void updateEdgeScore(XMapNode node1, XMapNode node2,
                                 XMapTDG graph, double scoreChange) {
        AssociationEdge edge;

//      System.out.println("Updating this edgeScore");
//      System.out.println(node1.getName() + "->" + node2.getName() +
//                         "\t" + scoreChange);
        try {
            if (graph.hasAssociation(node1, node2)) {
                edge = (AssociationEdge) graph.getEdge(node1, node2);
                // Scores accumulate.
                // Need the next line due to roundoff-type errors
                if (edge.getScore() < 1.0)
                    edge.setScore(combineScores(edge.getScore(), scoreChange));
            } else {
                graph.addEdge(node1, node2,
                              new AssociationEdge(node1, node2, 0,
                                                  scoreChange));
            }
        } catch (DuplicateEdgeException e) {
            // Means there is already an edge representing e.g.
            // a heirarchy.  Don't need to do anything.
        } catch (NoEdgeException e) {
            e.printStackTrace();
            // Thrown by TaggedDiGraph.getEdge,
            //  -- Shouldn't happen
        } catch (NoNodeException e) {
            e.printStackTrace();
            // Thrown by TDG.addEdge  -- Shouldn't happen
        }
    }


    /* Scores nodes as per their proximity to known associations.  This
     * way, we can speculate an association between nodes that are
     * hierarchically connected to associated nodes.
     */
    private UniqueHashMap scoreNodes(XMapNode scoreFromNode, XMapTDG graph,
                                     XMapNode otherNode) {
        /** Traverses the graph beginning at startingNode, and returns a
         * UHM of nodes scored by their proximity to startingNode.  The
         * score falls off exponentially as scoreDropOff, and the graph is
         * traversed checkLevels deep.  If two or more paths exist to a
         * particular node, the shortest path is counted.  Direction of
         * edges is not taken into account.
         */
        UniqueHashMap scoredNodes = new UniqueHashMap();
        HashSet nodesThisLevel = new HashSet();
        nodesThisLevel.add(scoreFromNode);
        HashSet nodesNextLevel;
        Iterator iterThisLevel;
        XMapNode node = null;
        double currentScore = 1.0;

//      try {
//          AssociationEdge thisEdge = (AssociationEdge)
//              graph.getEdge(scoreFromNode, otherNode);
//          System.out.println("\n\nGot to scoreNodes");
//          thisEdge.printEdge();
//      } catch(NoEdgeException e) {
//      }

        for (int j=0; j<checkLevels; j++) {
            nodesNextLevel = new HashSet();
            iterThisLevel = nodesThisLevel.iterator();
            while (iterThisLevel.hasNext()) {
                try {
                    node = (XMapNode) iterThisLevel.next();

                    if (!node.sameDomain(otherNode)) {
                        scoredNodes.put(node, new Double(currentScore));
                        nodesNextLevel.addAll(graph.hierarchySuccessors(node));
                        nodesNextLevel.addAll(graph.confidentMappings(node));
                        // Can look up the tree too.
                        //  these next 2 lines must be modified if not using trees.
                        if (node.getParent() != null)
                            nodesNextLevel.add(node.getParent());
                    }
                } catch (DuplicateException ex) {
                    // node already exists in scoredNodes,
                    // Nothing needs to be done.
                } catch (NoSuchElementException ex) { // Shouldn't happen
                } catch (NoNodeException ex) { // Shouldn't happen
                }
                nodesThisLevel = nodesNextLevel;
            }
            currentScore *= scoreDropOff;
        }

        // don't keep original node
        try {
            scoredNodes.remove(scoreFromNode);
        } catch (NotFoundException e) {
            e.printStackTrace(); //Shouldn't happen
        }
        return scoredNodes;
    }

    /* A function that combines scores that accrue from two distinct events
     * (such as nearby known associations)
     */
    private double combineScores(double d1, double d2) {
        // Discussed characteristics of this function with Mike.
        // Using below for the moment.

        return d1 + d2 - d1*d2;
    }

    /**
     * Sets the checkLevels parameter
     * @param cl The new number of levels up and down to check
     */
    public void setCheckLevels(int cl)
    {
        checkLevels = cl;
    }

    /**
     * Sets the scoreDropOff parameter
     * @param sdo The new amount by which score drops off with distance
     */
    public void setScoreDropOff(double sdo)
    {
        scoreDropOff = sdo;
    }

    /**
     * Sets the scoreThreshold parameter
     * @param st The new threshold above which to discard associations
     * as insignificant
     */
    public void setScoreThreshold(double st)
    {
        scoreThreshold = st;
    }
}
```

```
/*
 * TaggedDiGraph.java
 *
 * Revision History
 *   initial version - 06/09/2000: David Wang
 *   revised version - 07/22/2000: Eddie Byon
 *   revised version - 09/25/2000: Ashish Mishra
 *
 *
 * The contents of this file can be freely copied and used as long as the author
 * is credited either explicitly or implicitly by preserving this notice along
 * with any documentation in this file.
 */

package org.mitre.XMap;

import java.util.Iterator;
import java.util.ArrayList;
import java.util.NoSuchElementException;

import org.mitre.XMap.UniqueHashMap;
import org.mitre.XMap.NoNodeException;
import org.mitre.XMap.DuplicateNodeException;
import org.mitre.XMap.NoEdgeException;
import org.mitre.XMap.DuplicateEdgeException;
import org.mitre.XMap.Node;
import org.mitre.XMap.Edge;

/**
 * A TaggedDigraph is a mutable directed graph whose nodes and edges can be
 * tagged with arbitrary Objects.
 *
 * It is a pair <CODE>&lt;N, E&gt;</CODE>, where:
 * <ul>
 *     <li>N is the set of nodes in the graph
 *     <li>E is the set of edges between nodes in N
 * </ul>
 *
 * Each node is an Object.
 * An edge is <CODE>&lt;from-node, to-node, edge-tag&gt;</CODE>, where:
 * <ul>
 *     <li><CODE>from-node</CODE> and <CODE>to-node</CODE> are in N
 *     <li><CODE>tag</CODE> is an Object
 * </ul>
 *
 * There is at most one edge from any node to another node.
 *
 * @author David Wang
 * @author Eddie Byon
 * @version 1.0
 * @see org.mitre.XMap.XMap
 */
public class TaggedDiGraph {
    /**
     * HashMap within HashMap representation of a tagged, directed graph.
     */
    private UniqueHashMap taggedDG;

    /**
     * Constructs a new, empty tagged directed graph.
     *
     * @since 1.0
     */
    public TaggedDiGraph() {
        //effects:      Initializes this to be an empty digraph
        taggedDG = new UniqueHashMap();
    }

    /**
     * Adds the specified node as a vertex into this directed graph.
     * <P>
     * The node is required to not be null.
     * <P>
     *
     * @param node node to be added to this directed graph
     * @exception DuplicateNodeException if the node exists in the directed
     * graph
     * @since 1.0
     */
    public void addNode(Node mynode)
        throws DuplicateNodeException {
        try {
            taggedDG.put(mynode, new UniqueHashMap());
        }
        catch (DuplicateException e) {
            throw new DuplicateNodeException(mynode + " already exists");
        }
    }

    /**
     * Adds an edge from initialNode to finalNode with tag edgeTag.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     * <P>
     *
     * @param initialNode the starting node of the edge
     * @param finalNode the terminating node of the edge
     * @param edgeTag the tag associated with the edge
     * @exception NoNodeException if either initialNode or finalNode is not
     * in this graph
     * @exception DuplicateEdgeException if the edge already exists
     * @since 1.0
     */
    public void addEdge(Node initialNode, Node finalNode, EdgeTag edgeTag)
        throws NoNodeException, DuplicateEdgeException {

        addEdge(initialNode, finalNode, (Edge) edgeTag.getOwner());
    }

    /**
     * Adds a new edge from initialNode to finalNode.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     * <P>
     *
     * @param initialNode the starting node of the edge
     * @param finalNode the terminating node of the edge
     * @param edge the new edge added
     * @exception NoNodeException if either initialNode or finalNode is not
     * in this graph
     * @exception DuplicateEdgeException if the edge already exists
     * @since 1.0
     */
    public void addEdge(Node initialNode, Node finalNode, Edge edge)
        throws NoNodeException, DuplicateEdgeException {

        try {
            UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
            taggedDG.get(finalNode); // check that this node is also in digraph
            toNodes.put(finalNode, edge);
        } catch (NotFoundException e) {
            throw new NoNodeException(initialNode + " and/or " +
                                      finalNode + " does not exist");
        } catch (DuplicateException ee) {
            throw new DuplicateEdgeException
                ("An edge already exists between " +
                 initialNode + " and " + finalNode);
        }
    }

    /**
     * Verifies whether initialNode and finalNode exist in the graph,
     * and an edge exists from initialNode to finalNode.
     * Returns true or false depending on whether the edge exists.
     * <P>
     * initialNode is required to not be null.
     * <P>
     *
     * @param initialNode the starting node of the edge
     * @param finalNode the terminating node of the edge
     */
    public boolean hasEdge(Node initialNode, Node finalNode) {
        try {
            taggedDG.get(finalNode);
            UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
            toNodes.get(finalNode);
        } catch (NotFoundException e) {
            return false;
        }
        return true;
    }

    /**
     * Verifies whether thisNode exists in the graph.
     * Returns true or false depending on whether the node exists.
     * <P>
     * thisNode is required to not be null.
     * <P>
     *
     * @param thisNode the node to be verified
     * @since 1.0
     */
    public boolean hasNode(Node thisNode) {
                return taggedDG.containsKey(thisNode);
    }

    /**
     * Adds newNode and an edge of edgeTag from initialNode to newNode in this
     * graph.
     * <P>
     * The initialNode, newNode, and edgeTag are required to not be null.
     *
     * @param initialNode the starting node of the edge
     * @param newNode the new terminating node of the edge
     * @param edgeTag the tag associated with the edge
     * @exception NoNodeException if initialNode is not in this graph
     * @exception DuplicateNodeException if newNode already exists in this graph
     * already exists in this graph
     * @since 1.0
     */
    public void addNodeAndEdge(Node initialNode, Node newNode, EdgeTag edgeTag)
        throws NoNodeException, DuplicateNodeException {
        addNodeAndEdge(initialNode, newNode, (Edge) edgeTag.getOwner());
        //      try {
        //          UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
        //          toNodes.put(newNode, edgeTag);
        //      } catch (NotFoundException e) {
        //          throw new NoNodeException(initialNode + " does not exist");
        //      } catch (DuplicateException ee) {
        //          throw new DuplicateNodeException(newNode + " already exists");
        //      }
    }

    /**
     * Adds newNode and an edge from initialNode to newNode in this
     * graph.
     * <P>
     * The initialNode, newNode, and edge are required to not be null.
     *
     * @param initialNode the starting node of the edge
     * @param newNode the new terminating node of the edge
     * @param edgeTag the tag associated with the edge
     * @exception NoNodeException if initialNode is not in this graph
     * @exception DuplicateNodeException if newNode already exists in this graph
     * already exists in this graph
     * @since 1.0
     */
    public void addNodeAndEdge(Node initialNode, Node newNode, Edge edge)
        throws NoNodeException, DuplicateNodeException {
        addNode(newNode);
        try {
            addEdge(initialNode, newNode, edge);
        } catch (DuplicateEdgeException e) {
            // can't happen since we just added newNode
        }

        //      try {
        //          UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
        //          toNodes.put(newNode, edgeTag);
        //      } catch (NotFoundException e) {
        //          throw new NoNodeException(initialNode + " does not exist");
        //      } catch (DuplicateException ee) {
        //          throw new DuplicateNodeException(newNode + " already exists");
        //      }
    }

    /**
     * Removes the specified edge between the given nodes.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @exception NoEdgeException if initialNode, finalNode, or the edge from
     * initialNode to finalNode does not exist in this graph
     * @since 1.0
     */
    public void removeEdge(Node initialNode, Node finalNode)
        throws NoEdgeException {
        try {
            UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
            toNodes.remove(finalNode);
        } catch (NotFoundException e) {
            throw new NoEdgeException("Edge does not exist between " +
                                      initialNode.getName()
                                      + " and " + finalNode.getName());
        }
    }

    /**
     * Removes the specified edge from the graph.
     * <P>
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @exception NoEdgeException if the edge does not exist in the graph
     * @since 1.0
     */
    public void removeEdge(Edge edge)
        throws NoEdgeException {

        Node from = edge.getFromNode();
        Node to = edge.getToNode();
        try {
            // make sure this is actually the edge connecting these two
            if (getEdge(from, to) == edge)
                removeEdge(from, to);
        } catch (NoEdgeException e) {
            throw new NoEdgeException("This edge does not exist within the graph.");
        }
    }

    /**
     * Changes the tag of the specified edge between the given nodes to edgeTag.
     * <P>
     * The initialNode, finalNode, and edgeTag are required to not be null.
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @param edgeTag the new Tag value for this edge
     * @exception NoEdgeException if initialNode, finalNode, or the edge from
     * initialNode to finalNode does not exist in this graph
     * @since 1.0
     */
    public void changeTag(Node initialNode, Node finalNode, EdgeTag edgeTag)
        throws NoEdgeException {
```

```
        try {
            UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
            Edge edge = (Edge)toNodes.get(finalNode);
            edge.setEdgeTag(edgeTag);
            //      toNodes.change(finalNode, edgeTag);
        } catch (NotFoundException e) {
            throw new NoEdgeException("Edge does not exist between " +
                                      initialNode + " and " + finalNode);
        }
    }

    /**
     * Returns the tag of the specified edge between the given nodes.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @return the Tag of the directed edge from initialNode to finalNode
     * @exception NoEdgeException if initialNode, finalNode, or the edge from
     * initialNode to finalNode does not exist in this graph
     * @since 1.0
     */
    public EdgeTag edgeTag(Node initialNode, Node finalNode)
        throws NoEdgeException {
        return getEdge(initialNode, finalNode).getEdgeTag();
    }

    /**
     * Returns the edge connecting the given nodes.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @return the directed edge from initialNode to finalNode
     * @exception NoEdgeException if initialNode, finalNode, or the edge from
     * initialNode to finalNode does not exist in this graph
     * @since 1.0
     */
    public Edge getEdge(Node initialNode, Node finalNode)
        throws NoEdgeException {

        try {
            UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
            return (Edge)toNodes.get(finalNode);
            //      et = (EdgeTag)toNodes.get(finalNode);
            //      return (Edge) et.getOwner();
        } catch (NotFoundException e) {
            throw new NoEdgeException("Edge does not exist between " +
                                      initialNode + " and " + finalNode);
        }
    }

    /**
     * Returns the inverse of the edge connecting the given nodes.
     * i.e., the edge from the destination node of this edge to its
     * source node.
     * <P>
     * Both initialNode and finalNode are required to not be null.
     *
     * @param edge the directed edge who&quot;s inverse is required.
     * @return the directed edge in the opposite direction, if it exists,
     * otherwise null
     * @exception NoEdgeException if edge does not exist in this graph
     */
    public Edge inverseEdge(Edge edge)
        throws NoEdgeException {

        // Verify that edge is in the graph.
        try {
            if (edge != getEdge(edge.getFromNode(),
                                edge.getToNode())) {
                throw new NoEdgeException("Edge does not exist " +
                                           "in this graph");
            }
        } catch(NoEdgeException e) {
            throw new NoEdgeException("Edge does not exist " +
                                     "in this graph");
        }
        // edge is in the graph, now return the inverse
        try {
            return getEdge(edge.getToNode(),
                           edge.getFromNode());
        } catch (NoEdgeException e) {
            return null;
        }
    }

//    /**
//     * Allows one to add an edge between initialNode and finalNode without
//     * passing in an Edge.  Instead, a new Edge is created and newString
//     * is set as the EdgeTag String
//     *
//     * @param initialNode the originating node of the directed edge
//     * @param finalNode the destinating node of the directed edge
//     * @param newString the EdgeTag data of the directed edge
//     * @exception NoEdgeException if initialNode, finalNode, or the edge from
//     * initialNode to finalNode does not exist in this graph
//     * @since 1.0
//     */
//    public void addEdge(Node initialNode, Node finalNode, String newString)
//     throws DuplicateEdgeException, NoNodeException {
//     try {
//         Edge edge = new Edge(initialNode, finalNode);
//         EdgeTag newEdgeTag = new EdgeTag(edge, newString);
//         edge.setEdgeTag(newEdgeTag);

//         taggedDG.get(finalNode); // make sure it's there
//         UniqueHashMap toNodes = (UniqueHashMap)taggedDG.get(initialNode);
//         toNodes.put(finalNode, edge);
//         //      toNodes.put(finalNode, newEdgeTag);
//     } catch (NotFoundException e) {
//         throw new NoNodeException(initialNode + " and/or " +
//                                   finalNode + " does not exist");
//     } catch (DuplicateException ee) {
//         throw new DuplicateEdgeException
//             ("An edge already exists between " +
//              initialNode + " and " + finalNode);
//     }
//    }

    /**
     * Changes the EdgeTag data of an edge between initialNode and finalNode
     * to contain newString.
     *
     * @param initialNode the originating node of the directed edge
     * @param finalNode the destinating node of the directed edge
     * @param newString the EdgeTag data to be changed to.
     * @exception NoEdgeException if initialNode, finalNode, or the edge from
     * initialNode to finalNode does not exist in this graph
     * @since 1.0
     */
    public void changeEdgeTagString(Node initialNode, Node finalNode,
                                    String newString)
        throws NoEdgeException {
        EdgeTag edgeTag = edgeTag(initialNode, finalNode);
        edgeTag.setTagString(newString);
    }

    /**
     * Returns an Iterator that will produce each node in this graph exactly
     * once, in arbitrary order.
     * <P>
     * @return an Iterator that produces each node in this graph exactly once,
     * in arbitrary order
     * @since 1.0
     */
    public Iterator nodes() {
        return(taggedDG.keySet().iterator());
    }

    /**
     * Returns an Iterator that will produce each edge in this graph exactly
     * once, in arbitrary order.
     * <P>
     *
     * @return an Iterator that produces each edge in this graph exactly once,
     * in arbitrary order
     * @since 1.0
     */
    public Iterator edges() {
        //              return new EdgeIterator();
        ArrayList edges = new ArrayList();

        Iterator fromNodes, successorEdges;
        Node fromNode = null, toNode = null;

        fromNodes = nodes();
        try {
            while (fromNodes.hasNext()) {
                fromNode = (Node) fromNodes.next();
                // Add all successor edges for fromNode
                edges.addAll(((UniqueHashMap)taggedDG.get(fromNode)).values());
            }
        } catch (NoSuchElementException e) {
        }
        return edges.iterator();
    }

    /**
     * Returns an ArrayList containing the successors (the nodes) that
     * the given initialNode points to exactly once, in arbitrary order.
     * <P>
     * The initialNode is required to not be null.
     *
     * @param initialNode the node whose successors (nodes that are pointed to)
     * will be produced
     * @return an ArrayList that produces each successor of the given node
     * exactly once, in arbitrary order
     * @exception NoNodeException if initialNode does not exist in this graph
     * @since 1.0
     */
    public ArrayList successors(Node initialNode)
        throws NoNodeException {

        try {
            ArrayList arrayTaggedDG = new ArrayList
                (((UniqueHashMap)taggedDG.get(initialNode)).keySet());
            return arrayTaggedDG;
        }
        catch (NotFoundException e) {
            throw new NoNodeException(initialNode + " does not exist");
        }
    }

    /**
     * Returns an ArrayList containing the successor edges (not nodes) that
     * the given initialNode points to exactly once, in arbitrary order.
     * <P>
     * The initialNode is required to not be null.
     *
     * @param initialNode the node whose successors (nodes that are pointed to)
     * will be produced
     * @return an ArrayList that produces each successor of the given node
     * exactly once, in arbitrary order
     * @exception NoNodeException if initialNode does not exist in this graph
     * @since 1.0
     */
    public ArrayList successorEdges(Node initialNode)
        throws NoNodeException {
```

```
        try {
            ArrayList arraySE = new ArrayList
                (((UniqueHashMap)taggedDG.get(initialNode)).values());
            return arraySE;
        }
        catch (NotFoundException e) {
            throw new NoNodeException(initialNode + " does not exist");
        }
    }

    /**
     * Returns a string representation of the directed graph.
     *
     * @return a string representation of the directed graph
     * @since 1.0
     */
    public String toString() {
        return(taggedDG.toString());
    }


    /* All of the following commented code was previously used by the edges()
     * method.  We decided to shift over to an implementation that was
     * conceptually simpler and more elegant.  It is less efficient in storage,
     * but since the Iterator only contains pointers to Edges it shouldn't be
     * too bad.  Also, efficiency is not a primary consideration at this point,
     * and the edges will (relatively) rarely be iterated.
     */
//      /**
//       * An EdgeIterator is a lazy-eval Iterator for the edges contained in its
//       * associated directed graph.
//       *
//       * @author David Wang
//       * @version 1.0
//       * @see org.mitre.XMap.TaggedDiGraph
//       */
//          class EdgeIterator implements Iterator {
//                  /*
//                   * BUGBUG: fail-fast if edges/nodes are added while in use
//                   *
//                   * In an attempt to optimize spaces used while iterating, we only hold
//                   * onto an Iterator of the nodes in this graph.  This means that we
//                   * cannot detect and fail-fast if an edge was added between two existing
//                   * nodes since no nodes were added/removed.
//                   *
//                   * One way to deal could be to have pointers to Iterators of all
//                   * UniqueHashMaps in the graph.  However, this makes the creation
//                   * of an EdgeIterator to be expensive (it essentially creates the whole
//                   * graph, meaning space consumption is 2X).
//                   *
//                   * The current implementation only makes it a requirement that the
//                   * graph be not modified while in use if it is to be considered
//                   * correct.
//                   */

//                  /** The current ''from'' node being iterated over. */
//                  private Node fromNode;

//                  /** All of the originating nodes in the directed graph. */
//                  private Iterator fromNodes;

//                  /** All of the destination nodes of the currint ''from'' node. */
//                  private Iterator toNodes;

//                  /**
//                   * Constructs a new Iterator based on its encompassing directed graph.
//                   *
//                   * @since 1.0
//                   */
//                  EdgeIterator() {
//                          /*
//                           * If fromNodes is empty, toNodes is also empty; else, toNodes is
//                           * the ''to'' nodes of the first from node.
```

```
    //                          */
    //                          fromNodes = nodes();
    //                          toNodes = getNextToNodes(fromNodes);
    //                          /*
    //                            toNodes = fromNodes.hasNext() ?
    //                            successors(fromNodes.next()) :
    //                            new UniqueHashMap().keySet().iterator();
    //                          */
    //                  }

    //                  /**
    //                   * Returns true if the Iterator has more elements. (in other words,
    //                   * returns true if next would return an element rather than throwing an
    //                   * exception.)
    //                   *
    //                   * @return true if the Iterator has more elements; false otherwise
    //                   * @since 1.0
    //                   */
    //                  public boolean hasNext() {
    //                          /*
    //                           * If fromNodes is empty, toNodes is also empty and returns false
.
    //                           * If fromNodes is not empty, toNodes contains whatever nodes com
e
    //                           * from the current from node and hence is the same as what's nex
t
    //                           */
    //                          return fromNodes.hasNext();
    //                  }

    //                  /**
    //                   * Returns the next element in the interation.
    //                   *
    //                   * @exception NoSuchElementException if the Iterator has no more
    //                   * elements
    //                   * @since 1.0
    //                   */
    //                  public Object next() {
    //                          /*
    //                           * If the toNodes hasNext, return it.  If it does not, then
    //                           * increment fromNodes and try again
    //                           */
    //                          try {
    //                                  if (!toNodes.hasNext()) {
    //                                          toNodes = getNextToNodes(fromNodes);
    //                                  }

    //                                  Node toNode = (Node)toNodes.next();
    //                                  try {
    //                                          return new Edge(fromNode,
    //                                                                          toNode,
    //                                                                          (EdgeTag)edgeTag(
fromNode, toNode));
    //                                  } catch (Exception e) {
    //                                          //catch NoEdgeException, NoNodeException, ClassCa
stException
    //                                          Assert.assert(false);  //this should never be the
 case

    //                                          return null;  //appease javac
    //                                  }
    //                          } catch (ClassCastException ee) {
    //                                  Assert.assert(false);  //this should never be the case

    //                                  return null;  //appease javac
    //                          }
    //                  }

    //                  /**
    //                   * Returns an Iterator of ``to'' nodes of the next ``from'' node of
    //                   * the given Iterator.  If there is no next ``from'' node, return an
    //                   * empty Iterator.
    //                   *
    //                   * @return an Iterator of ``to'' nodes of the next ``from'' node
    //                   * @since 1.0
    //                   */
    //                  private Iterator getNextToNodes(Iterator fromNodes) {
    //                          /* If fromNodes is empty, toNodes will also return an empty
    //                           * Iterator.  If it is not empty, then toNodes will be the ''to''
    //                           * nodes of the next from node.
    //                           */
    //                          try {
    //                                  return fromNodes.hasNext() ?
    //                                          successors(fromNode = (Node)fromNodes.next()) :
    //                                          new UniqueHashMap().keySet().iterator();
    //                          } catch (NoNodeException e) {
    //                                  Assert.assert(false);  //this should never be the case

    //                                  return null;  //appease javac
    //                          } catch (ClassCastException ee) {
    //                                  Assert.assert(false);  //this should never be the case

    //                                  return null;  //appease javac
    //                          }
    //                  }

    //                  /**
    //                   * Removes from the underlying collection the last element returned by
    //                   * the Iterator (optional operation). This method can be called only
    //                   * once per call to next.
    //                   * <P>
    //                   * The behavior of an Iterator is unspecified if the underlying
    //                   * collection is modified while the Iterator is Node test1=(Node)test.nex
t();in progress in any way
    //                   * other than by calling this method.
    //                   *
    //                   * @exception UnsupportedOperationException if the remove operation is
    //                   * not supported by this Iterator
    //                   * @exception IllegalStateException if the next method has not yet been
    //                   * called, or the remove method has already been called after the last
    //                   * call to the next method
    //                   * @since 1.0
    //                   */
    //                  public void remove() {
    //                          toNodes.remove();
    //                  }
    //          }

    /**
     * Contains static test cases for this method.  Do not invoke casually.
     *
     * @since 1.0
     */
    public static void selfTest() {
        TaggedDiGraph testGraph1 = new TaggedDiGraph();

        System.out.println(testGraph1.getClass().toString());
    }
}
```

```
package org.mitre.XMap;

import java.io.FileNotFoundException;
//  import javax.swing.JFrame;
//  import java.awt.event.WindowAdapter;
//  import java.awt.event.WindowEvent;

import java.util.Iterator;
import java.util.ArrayList;
import java.util.NoSuchElementException;

import org.mitre.XMap.XMapTDG;
import org.mitre.XMap.XML2DiGraph;
import org.mitre.XMap.XMapNode;
import org.mitre.XMap.LinkBase;
import org.mitre.XMap.StructuralAnalysis1;
import org.mitre.XMap.EquivalenceAnalysis;


public class XMapGT {
    public static void main(String[] args) {
        // For now, just two file (to be changed)
        //      int numFiles = args.length / 2;
        int numFiles = 2;
        XML2DiGraph x2dg = new XML2DiGraph();
        XMapTDG tdg = new XMapTDG();
        if(numFiles < 1) {
            System.err.println("Need to specify XML File and Element to print from");
            System.exit(-1);
        }

        for (int i=0; i<numFiles; i++) {

            String fileName = args[2*i];
            String printFromElement = args[2*i+1];

            try {
                x2dg.xml2Graph(fileName, tdg, fileName);
                System.out.println("Read " + fileName);
                                //x2dg.linkbase2Graph(fileName, tdg, fileName);
            } catch(FileNotFoundException e) {
                System.err.println("File " + fileName + " not found");
                System.exit(-1);
            }

            Iterator nodes1 = tdg.nodes();
//          XMapNode thisNode=null;
//              //          System.out.println("Tree from element with name " + printFromElement)
;

//          while(nodes1.hasNext()) {
//              try {
//                  thisNode = (XMapNode)nodes1.next();
//                  if(thisNode.getName().equalsIgnoreCase(printFromElement)) {
//                          //tdg.printTreeFromNode(thisNode, "");
//                  }
//              } catch (NoSuchElementException e) {
//                  e.printStackTrace();
//              }
//          }
        }

//      Linkbase2DiGraph l2dg = new Linkbase2DiGraph();
//      String fileName = "XMLLinkbase.xml";
//      String printFromElement = "Mappings";

//      try {
//          l2dg.linkbase2Graph(fileName, tdg, fileName);
//      } catch(FileNotFoundException e) {}

        Iterator nodes1;
        XMapNode thisNode, tempNode;

        EquivalenceAnalysis myEA=new EquivalenceAnalysis();
        DataAnalysis myDA=new DataAnalysis();
        nodes1=tdg.nodes();
        XMapNode root1=null;
        XMapNode root2=null;
        Iterator domain1Nodes, domain2Nodes;
        try {
            while(nodes1.hasNext()) {
                thisNode = (XMapNode)nodes1.next();
                if(thisNode.getName().equalsIgnoreCase(args[1]))
                    root1 = thisNode;
                if(thisNode.getName().equalsIgnoreCase(args[3]))
                    root2 = thisNode;
            }
            String root1Domain = (String)root1.getDomain();
            String root2Domain = (String)root2.getDomain();
            String lookFor = "";
            domain1Nodes = tdg.nodes();
            while (domain1Nodes.hasNext()) {
                domain2Nodes = tdg.nodes();
                tempNode = (XMapNode)domain1Nodes.next();
                if (!tempNode.sameDomain(root1) && !tempNode.sameDomain(root2)) {
                    //do nothing
                } else {
                    if (tempNode.sameDomain(root1))
                        lookFor = root2Domain;
                    else
                        lookFor = root1Domain;
                    while (domain2Nodes.hasNext()) {
                        XMapNode tempNode2 = (XMapNode)domain2Nodes.next();
                        if (((String)tempNode2.getDomain()).equals(lookFor)) {
                            if (!tdg.hasEdge(tempNode, tempNode2)) {
                                myEA.equivalence(tempNode, tempNode2, tdg);
                                if (!tempNode.getData().isEmpty() && !tempNode2.getData().isEmpty
()) {
                                    double testDATA = myDA.data(tempNode, tempNode2, tdg);
                                    if (testDATA > 0.8) {
                                        testDATA=testDATA-0.5;
                                        if (!tdg.hasEdge(tempNode, tempNode2)) {
                                            AssociationEdge newAssociation = new AssociationEdge(
tempNode, tempNode2, 1);

                                            newAssociation.setDataScore(testDATA);
                                            try {
                                                tdg.addEdge(tempNode, tempNode2, newAssociation);
                                            }
                                            catch (DuplicateEdgeException e) {
                                                // Shouldn't happen
                                            }
                                        }
                                        else {
                                            AssociationEdge workingEdge = (AssociationEdge)tdg.ge
tEdge(tempNode, tempNode2);

                                            workingEdge.setDataScore(testDATA);
                                        }

                                        //
       System.out.print(" ");
                                        //
       System.out.print(testDATA);
                                        //
       System.out.print(" ");
                                        //
       System.out.print(tempNode.getName());
                                        //
       System.out.print(" --- ");
                                        //
       System.out.println(tempNode2.getName());
                                    }
                                }
                            }
                        }
                    }
                }
            }
            //UNCOMMENT THIS IF YOU WANT TO SEE ASSOCIATION EDGES

//          System.out.println("Association Edges present after Equals Analysis");
//          Iterator associations = tdg.associationEdges().iterator();
//          while (associations.hasNext()) {
//              AssociationEdge testingEdge = (AssociationEdge)associations.next();
//              testingEdge.printEdge();
//          }
//          System.out.println();

            Iterator associations = tdg.associationEdges().iterator();
            while (associations.hasNext()) {
                AssociationEdge testingEdge = (AssociationEdge)associations.next();
                double score = testingEdge.getScore();
                AssociationEdge upgradeEdge = null;
                XMapNode parentFromNode = null;
                XMapNode parentToNode = null;
                if (((XMapNode)testingEdge.getFromNode()).isAttribute() || ((XMapNode)testingEdge
.getToNode()).isAttribute()) {
                    parentFromNode = ((XMapNode)testingEdge.getFromNode()).getParent();
                    parentToNode = ((XMapNode)testingEdge.getToNode()).getParent();
                    if (tdg.hasAssociation(parentFromNode, parentToNode)) {
                        upgradeEdge=(AssociationEdge)tdg.getEdge(parentFromNode, parentToNode);
                        System.out.print(upgradeEdge.getScore());
                        upgradeEdge.updateScore();
                        System.out.print(" <- old score | new score ->");
                        System.out.println(upgradeEdge.getScore());

                    } else {
                        AssociationEdge newEdge = new AssociationEdge(parentFromNode, parentToNod
e, 0, score);
                        try {
                            tdg.addEdge(parentFromNode, parentToNode, newEdge);
                        }
                        catch (DuplicateEdgeException e) {
                            System.out.println("Edge already exists.");
                                        }
                                        newEdge.printEdge();
                                }
                        }
                }
        }
        catch (Exception e) {
            e.printStackTrace();
        }

//      //       UNCOMMENT THIS IF YOU WANT TO SEE DATA LIST
//              Iterator testData = tdg.nodes();
//              while (testData.hasNext()) {
//      //          int i=0;
//                  tempNode=(XMapNode)testData.next();
//                          }

        //              System.out.println(tempNode.getName());
        //              ArrayList tempData=tempNode.getData();
        //              for(i=0;i<tempData.size();i++ ) {
        //                  System.out.print("  ");
        //                  System.out.println(tempData.get(i));
        //                  //          }
        //          }
        //      }

        SA2 mySA;

//      if (args.length == 7) {
//          try {
//              int cl = Integer.parseInt(args[4]);
//              double sdo = Double.parseDouble(args[5]);
//              double st = Double.parseDouble(args[6]);
//              mySA = new StructuralAnalysis1(cl, sdo, st, root1, root2);
//          } catch (NumberFormatException e) {
//              System.err.println("Couldn't parse arguments as SA parameters.");
//              System.err.println("Using default parameters.");
//              mySA = new StructuralAnalysis1(root1, root2);
//          }
//      }
//      else
            mySA = new SA2(root1, root2);

        mySA.analyze(tdg);

        Iterator associations = tdg.associationEdges().iterator();
        while (associations.hasNext()) {
                AssociationEdge testingEdge = (AssociationEdge)associations.next();
                System.out.print(testingEdge.mappingType());
                System.out.print(" ");
                testingEdge.printEdge();
        }

        UserQuery query = new UserQuery(tdg.associationEdges(),
                                        root1.getDomain(),
                                        root2.getDomain(),
                                        tdg, root1, root2);
    }
}
```

Bibliography

[1] P.V. Biron, A. Malhotra, editors. XML Schema Part 2: Datatypes. W3C Working Draft, April 7, 2000. http://www.w3.org/TR/xmlschema-2

[2] T. Bray, J. Paoli, C.M. Sperberg-McQueen, editors. eXtensible Markup Language (XML) 1.0. W3C Recommendation, February 10, 1998. http://www.w3.org/TR/REC-xml

[3] J. Clark, S. DeRose, editors. XML Path Language (XPath) Version 1.0. W3C Recommendation, November 16, 1999. http://www.w3.org/TR/xpath

[4] D.C. Fallside, editor. XML Schema Part 0: Primer. W3C Working Draft, April 7, 2000. http://www.w3.org/TR/xmlschema-0

[5] Ashish Mishra, Michael Ripley, Amar Gupta. Using Structural Analysis to Mediate XML Semantic Interoperability. MIT Sloan Working Paper No. 4345-02, February 2002. http://papers.ssrn.com/sol3/papers.cfm?abstract_id=306845

[6] C.H. Goh. Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems. PhD dissertation. Massachusetts Institute of Technology, Sloan School of Management, December 1996.

[7] C.F. Goldfarb, W.E. Kimber, P.J. Newcomb, S.R. Newcomb, editors. ISO/IEC 10744:1997 Architectural Form. August 27, 1997. http://www.ornl.gov/sgml/wg8/docs/n1920/html/toc.html

[8] W. Kim and J. Seo (1991). Classifying schematic and data heterogeneity in multidatabase systems. IEEE Computer, 24(12):12-18.

[9] R. Krishnamurthy, W. Litwin, and W. Kent (1991). Language features for interoperability of databases with schematic discrepancies. In Proceedings of the ACM SIGMOD Conference, pages 40-49.

[10] D. Megginson. Structuring XML Documents. 1998. Prentice Hall.

[11] C.F. Naiman and A.M. Ouskel (1995). A classification of semantic conflicts in heterogeneous database systems. Journal of Organizational Computing, 5(2):167-193.

[12] Report on Information Management to Support the Warrior, SAB-TR-98-02. December 1998. http://afosr.sciencewise.com/afr/sab/any/text/any/afrtstud.htm

[13] http://diides.ncr.disa.mil/shade

[14] D. Brickley, R.V. Guha, editors. Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, March 27, 2000. http://www.w3.org/TR/rdf-schema

[15] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, editors. XML Schema Part 1: Structures. W3C Working Draft, April 7, 2000. http://www.w3.org/TR/xmlschema-1

[16] J. Clark, editor. eXtensible Stylesheet Language Transform (XSLT) Version 1.0. W3C Recommendation, November 16, 1999. http://www.w3.org/TR/xslt

[17] M. Biezunski, M. Bryan, S. Newcomb, editors. ISO/IEC FCD 13250:1999 Topic Map. April 19, 1999. http://www.ornl.gov/sgml/sc34/document/0058.htm

[18] Y.J. Breitbart and L.R. Tieman (1985). ADDS: Heterogeneous distributed database system. In F. Schreiber and W. Litwin, editors, Distributed Data Sharing Systems, pages 7-24. North Holland Publishing Co.

[19] P. Ion and R. Miner, editors. Mathematical Markup Language. W3C Proposed Recommendation. February 24, 1998. http://www.w3.org/TR/PR-math/

[20] A. Sheth and V. Kashyap (1992). So far (schematically) yet so near (semantically). In D.K. Hsiao, E.J. Neuhold, R. Sacks-Davis. Proceedings of the IFIP WG2.6 Database Semantics Conference on Interoperable Database Systems (DS-5), pages 283-312, Lorne, Victoria, Australis. North-Holland.

[21] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37:237–240, February 1991.

[22] Claude Berge. *Graphs and Hypergraphs.* North-Holland Publishing Company, 1973.

[23] Norbert Blum. A new approach to maximum matching in general graphs. In *ICALP 90 Automata, Languages and Programming*, pages 586–597, Berlin, July 1990. Springer.

[24] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression, and speeding-up algorithms. In Baruch Awerbuch, editor, *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 123–133, New Orleans, LA, May 1991. ACM Press.

[25] Zvi Galil. Efficient algorithms for finding maximum matchings in graphs. *Computing Surveys*, 18:23–38, 1986.

[26] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4), December 1973.

[27] L. Lovász and M. D. Plummer. *Matching Theory*, volume 121 of *North-Holland mathematics studies*. North-Holland, Amsterdam, 1986.

[28] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximal matching in general graphs. In *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 17–27. IEEE Computer Society Press, 1980.

[29] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[30] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice Hall, 1982.

[31] Paul A. Peterson and Michael C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3:511–533, 1988.

[32] Andrew Shapira. Classes of graphs for which an $O(m + n)$ time greedy matching procedure does and does not find a maximum matching, in preparation.

[33] C. S.-M. T. Bray, J. Paoli and E. Maler. http://www.w3.org/TR/REC-xml, October 2000. eXtensible Markup Language (XML) 1.0, W3C recommendation.

[34] D. Wang. Automated semantic correlation between multiple schema for information exchange. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, 2000.

[35] E. Byon. X-Map Linkbase Search Engine (XLSE): A Search Engine for XML Documents. Draft Manuscript, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, 2002.

[36] JBI — Joint Battlespace Infosphere. http://www.rl.af.mil/programs/jbi/default.cfm, August 2001.

[37] Joint Battlespace Infosphere (JBI) Fuselet Research and Development. http://www.rl.af.mil/ div/IFK/prda/prda0009.html, January 2002.

[38] Protégé-2000. http://protege.stanford.edu/index.shtml, June 2001.

[39] The COntext INterchange Project http://context.mit.edu/~coin/, July 2000.

[40] C.F. Goldfarb, W.E. Kimber, P.J. Newcomb, S.R. Newcomb, editors. ISO/IEC 10744:1997 Architectural Form. http://www.ornl.gov/sgml/wg8/docs/n1920/html/toc.html, August 27, 1997.

[41] M. Biezunski, M. Bryan, S. Newcomb, editors. ISO/IEC FCD 13250:1999 Topic Map. http://www.ornl.gov/sgml/sc34/document/0058.htm, April 19, 1999.

[42] XML Linking Language (XLink) Version 1.0 http://www.w3.org/TR/xlink/, 27 June 2001

[43] M. M. Halldorsson and K. Tanaka. Approximation and special cases of common subtrees and editing distance. In *Proc. 7th Ann. Int. Symp. on Algorithms and Computation*, pages 75–84. Springer, 1996. Lecture Notes in Computer Science, number 1178.

[44] V. Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, 1992.

[45] V. Kann. On the approximability of the maximum common subgraph problem. In *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388. Springer, 1992. Lecture Notes in Computer Science, number 577.

[46] M. R. Garey, D. S. Johnson. Computers and Intractability: a guide to the theory of NP-completeness. W.H. Freeman and Company, San Francisco, 1979.

[47] C. Lin. Hardness of approximating graph transformation problem. In *Proc. 5th Ann. Int. Symp. on Algorithms and Computation*, pages 74–82. Springer, 1994. Lecture Notes in Computer Science, number 834.

[48] B. D. McKay. Nauty. http://cs.anu.edu.au/people/bdm/nauty/, February 2000.

[49] GMT - Graph Matching Toolkit http://www.cs.sunysb.edu/ algorith/implement/gmt/ implement.shtml, July 1999

[50] S. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory in Mathematica.* Addison-Wesley, Redwood City CA, June 1990.

[51] D. Kleitman, Personal Communication. MIT Mathematics Department (Combinatorics), June 2000.

[52] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *Inform. Process. Lett. 49, 249-254*, June 1994.

[53] Paul Erdös and T. Gallai. On maximal paths and circuits of graphs. *Acta Math. Acad. Sc. Hungar.*, 10:337–356, 1959.

[54] Andrew Shapira An Exact Performance Bound for an O(m+n) Time Greedy Matching Procedure Electronic Journal of Combinatorics Paper R25 of Volume 4(1)

[55] Andrew Shapira. Matchings, degrees, and $O(m + n)$ time procedures, in preparation.

[56] Wen-Syan Li and Chri Clifton. Semantic Integration in Heterogenous Databases using Neural Networks Proceedings of the 20th VLDB Conference, 1994

[57] Patrick Henry Winston Artificial Intelligence Addison-Wesley Co., 1992

[58] C. M. Sperberg-McQueen and Henry Thompson. XML Schema. W3C Architecture Domain, April 2000. http://www.w3.org/XML/Schema

[59] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, editors. XML Schema Part 1: Structures. W3C Recommendation, May 2, 2001. http://www.w3.org/TR/xmlschema-1

[60] P.V. Biron, A. Malhotra, editors. XML Schema Part 2: Datatypes. W3C Recommendation, May 2, 2001. http://www.w3.org/TR/xmlschema-2

[61] J. Clark, editor. eXtensible Stylesheet Language Transform (XSLT) Version 1.0. W3C Recommendation, November 16, 1999. http://www.w3.org/TR/xslt

[62] Francisco Rodriguez. Supervised and Unsupervised Neural Networks. November, 2001. http://www.gc.ssr.upm.es/inves/neural/ann1/concepts/Suunsupm.htm

[63] Tom Doris. JAva Neural Network Tool. July, 1999. http://www.compapp.dcu.ie/ tdoris/BCK/

[64] NIST. International System of Units. July 1999. http://physics.nist.gov/cuu/Units/index.html