A WORD NEXUS FOR SYSTEMATIC INTEROPERATION OF SEMANTICALLY HETEROGENEOUS DATA SOURCES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jan Frederic Jannink
March 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Gio Wiederhold (Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark Musen

Approved for the University Committee on Graduate Studies:

Abstract

Ever increasing amounts of information are available in digital form for use in existing and emerging applications. Data sources, formats and descriptions are accessible in a diversity unimaginable a few years ago. This information seldom comes with a complete specification or schema, even though much of it contains some regular structure. Existing specifications or ontologies, developed separately from the data, are of no direct benefit in organizing such volumes of data. Tools are needed to assist domain experts linking information from diverse and changing sources.

This dissertation presents the SKEIN system which is designed around an algebraic framework. SKEIN is a suite of tools for managing semantic heterogeneity between information sources. The presentation focuses on one large scale repository developed using the algebra. This repository, or nexus, is a graph of dictionary terms related by their definitions as extracted from an on-line Oxford English Dictionary resource. Two algorithms over the nexus provide assistance to experts in domain interoperation. ArcRank computes the most relevant arcs between terms, building on an extension of PageRank. All Pairs Similarity uses ArcRank values to compute which terms have the most similar link structure.

The nexus is a directed labeled graph, four times the size of two other lexical repositories, WordNet from Princeton U. and MindNet from Microsoft Research, but required orders of magnitude less development and maintenance effort. The operators used to build the repository are generic and apply equally well to thesauri, encyclopedias, and other dictionaries. The use of the nexus reduces the effort expended by the expert in matching terms between other sources. Given the task of pairing up English language pages of NATO government websites, SKEIN achieved 70% of the matches obtained by a human expert, without generating any false matches. The nexus and assorted algorithms, when used in the context of the SKEIN system, constitute the first steps towards the systematic interoperation of heterogeneous data sources.

Acknowledgments

More than most endeavors of this type, this work is the result of many people's efforts and patience. First and foremost, I thank my wife Dorothy, without whom none of it would have been possible. There is no doubt in my mind that her love was the fuel that rekindled my desire to complete this work. I owe a debt to everyone who believed in me when I wasn't sure I should believe in myself. Carolyn Tajnai and Suzanne Bentley at the Stanford Computer Forum helped me early on. Nice experiences, like the design of the T-shirt commemorating the 32$^{\mathrm{nd}}$ anniversary of the Computer Science Department, are a result of those contacts. Laura Haas at IBM Almaden Research lab gave me new confidence in my ability. The Garlic team provided a great environment to learn to enjoy coding again.

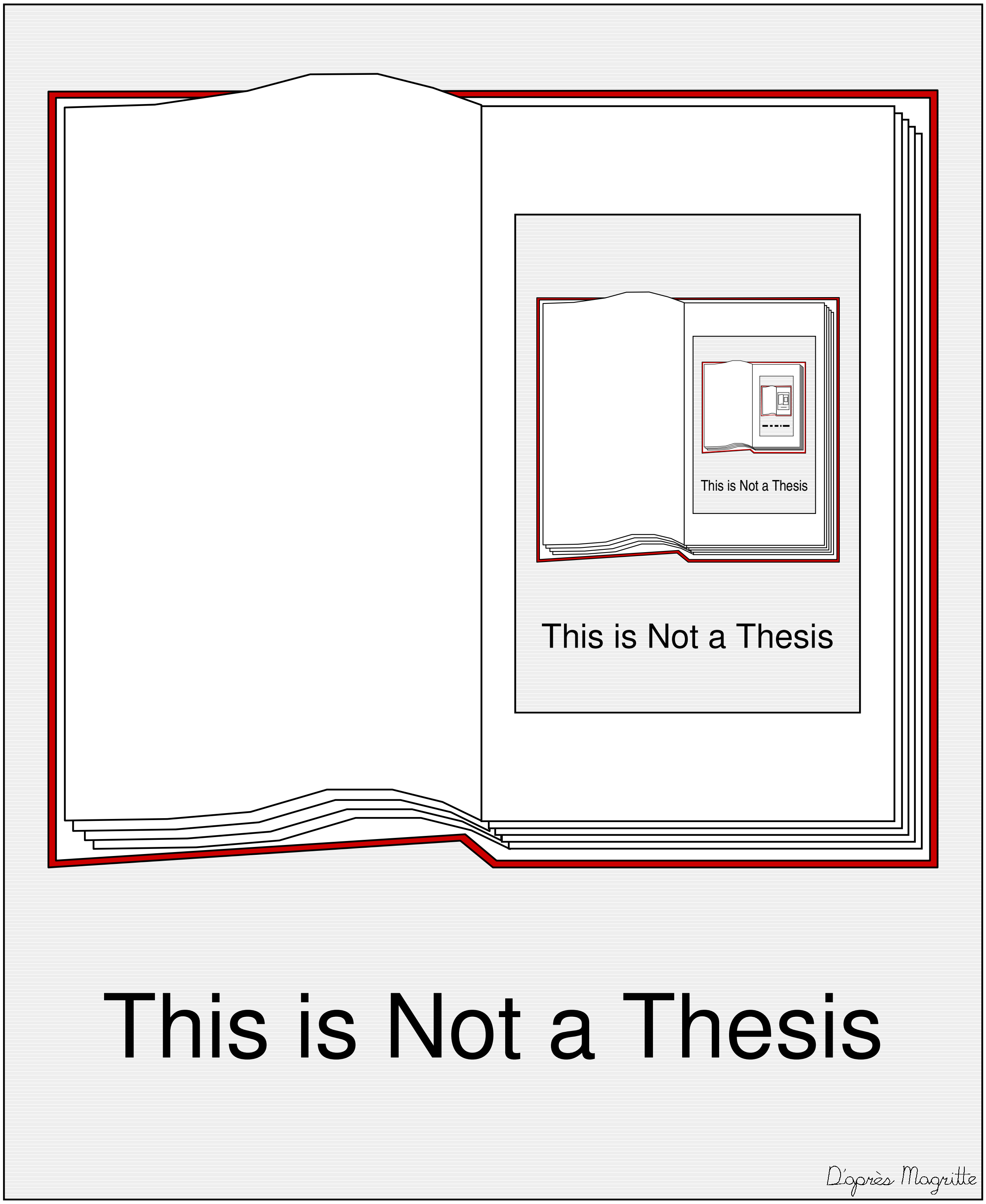
I thank Gio Wiederhold, my advisor, who gave me a great problem to study, and remarkable freedom to pursue various approaches to attack it. Professors Erich Neuhold and Rudi Studer, on sabbatical from Germany, contributed to the creative process. My fellow research group members Prasenjit Mitra, Vasan Pichai and Danladi Verheijen forced me to stop taking shortcuts when explaining ideas. Conversations with Mark Musen, Harold Boley and Martin Kersten sharpened the technical arguments. Interaction with Stefan Decker helped close gaps in my work. True friends, such as Narayanan Shivakumar, Luca de Alfaro, Sudarshan Chawathe, have helped me in both good and bad times. My friends at Gigabeat Inc. allowed me to develop my ideas in a commercial setting.

It is only fitting to remember the example set by my managers and coworkers at the Toshiba R & D center in Kawasaki, Japan, in particular, T. Kodama, T. Kamitake, and Y. Shobatake. It was their thorough knowledge of the networking field that first inspired me to consider graduate studies. I recognize my brother Jean-Luc, who just completed his own Ph.D., and from whom I learned the virtues of healthy competition. Last but not least, I thank my parents, who have always shown me love, and still provide moral support.

Contents

List of Tables

List of Figures


This is Not a Thesis
This is Not a Thesis
This is Not a Thesis
D'après Magritte

This is

Chapter 1

Introduction

The emergence of the World Wide Web as a universal medium of information exchange has radically transformed our ability to access and exploit heterogeneous information sources. This shift has exposed issues that have seldom met with deep inquiry in the context of database research. The assumptions about regularity and quality of information that go virtually unchallenged in traditional database research are demonstrably not valid in this environment. The fundamental problem we face is that information is always expressible and interpretable in multiple different ways. Despite this challenge, it is valuable to be able to exploit this wealth of information in a coherent fashion. In what follows we examine the relevance of semantic heterogeneity to data management, and we present a systematic approach to dealing with it.

Optical illusions and mirages tell us that our senses are not always capable of providing us with an accurate account of reality. Inspired by the work of Magritte, the preceding pages contain a lexical component which implicitly encourages us to question where reality ends and representation begins. Words are objects in their own right, but their principal purpose is to represent communication about other objects. *Semantic heterogeneity* arises out of the ambiguity inherent in the separation between words and what they represent. Puns, double entendres, and metaphors are the linguistic equivalents of optical illusions. These statements are examples of semantic heterogeneity, since they contain multiple meanings purposefully inserted into a single statement. Fortunately, in practice, we only have to deal with inadvertent semantic heterogeneity, that is, the cases where different meanings arise because of a lack of a pre-established common meaning. Recently the author had a first-hand experience with business errors caused by semantic heterogeneity on the Web. He received a targeted mass-mailing offering “the Visa platinum card exclusively for scuba divers,” despite the fact that his only affiliation to diving is the hobby of *springboard* diving. The author’s home page is the only public source of this information. Such *semantic mismatch* is by no means unusual. Most mass mailings also blindly apply the feminine gender to the first name Jan.

The database community is just beginning to investigate some of these issues in the context of *model management* [BLP00]. In the field of artificial intelligence, however, research has been accelerating in the last few years [JPVW98]. The methodology for representing a domain is named *ontology*, a term borrowed from philosophy. Ontology concerns itself with the representation of the objects in the universe and the web of their various connections. The traditional task of ontologists has been to extract from this tangle a single ordered structure, in the form of a tree or lattice [Sow00]. This structure consists of the terms that represent the objects, and the relationships that represent connections between objects.

We propose to defer the task of globally classifying terms and relationships, and to focus instead on identifying the appropriate ones as we need them. We distinguish *context* from concept, the unit of ontological abstraction, and *composition* from subsumption, or containment, the relationships which commonly provide structure to ontologies. Context expresses the conditions under which statements about concepts are true. Composition recognizes that statements from separate sources about the same objects are not a priori true in the same context. Composition also recognizes distinct objects as being equivalent in the right context. Whenever we compose information from separate contexts, we must create a new context to define the relationships between terms in the original contexts. As the composition of contexts requires the creation of new contexts we view this composition as an algebraic operation. We use *articulation* as the generic term for a context composition operation. Given the tools of context and composition, it now becomes possible to consider potential savings in the cost of constructing and maintaining ontologies from components, over the costs of monolithic ontologies.

This dissertation presents three results that show the value of the SKEIN system. As we mentioned in the abstract SKEIN, or Scalable Knowledge Extractor and Integrative Nexus, is a suite of tools, and the nexus is a repository of word relationships. The nexus we developed is four times larger than other comparable systems, but required orders of magnitude less development and maintenance effort. The operators used to build the nexus are generic and the articulations developed with them are reusable and scalable. The cost

Table 1.1: Information Sources

Source	Size [MB]
Oxford English Dictionary	570
Webster's Dictionary	50
Roget's Thesaurus	1
NATO & NATO governments' websites	10
British Airways Web schedule	6
United Airlines Web schedule	5

of applying the articulations grows linearly in the size of the information sources. The use of the nexus reduces the effort expended by the expert in matching terms between other heterogeneous sources, such as the pages of NATO government websites. We used a wide array of sources in the development of SKEIN and the nexus. Table 1.1 presents a list of the data sources that we discuss in the dissertation, along with their approximate raw size. Each of the first three sources listed can be used to build a nexus, and the other three sources serve to demonstrate the use of the SKEIN system. The three interdependent components of the SKEIN system, the word nexus and the algorithms computed over the nexus, constitute first steps towards the systematic interoperation of heterogeneous data sources.

The following introductory sections outline the material and contributions of the thesis along with related work. In Section 1.1 we illustrate with a motivating example the specific problems addressed in the dissertation. We also introduce the SKEIN system and the way we use it. We present our research hypotheses in Section 1.2 and briefly outline the form our results have taken. Section 1.3 defines the terms and their usage in this dissertation, thus the context of the thesis. We describe the solutions we have identified and implemented in Section 1.4, and present the related work in Section 1.5.

1.1 Semantic Interoperation

This section presents a definition of the problem we address in this work. We seek methods to achieve *efficient* identification of *relevant* information in heterogeneous repositories, *verification* of its appropriateness, its deployment and *maintenance* for *application specific* needs.

The major assumptions of this problem statement are the following:

- relevancy is defined with respect to application requirements
- a priori, no consistency exists between separate repositories
- data in individual sources may contain errors and irregularities
- sources may undergo autonomous changes at any time

More important than the assumptions are the consequences of the problem statement. In particular, efficiency drives us to avoid attempting to integrate the entire content of information sources. The need to maintain knowledge from independently changing sources means it is unproductive to convert more than the essential information from any source. Instead, we focus on achieving interoperation of the portions of sources relevant to a specific application. Another consequence of our problem statement is that we make no effort to process free text completely. Although we have performed partial text analysis in application specific ways, the problem of processing arbitrary texts is far beyond the scope of the dissertation. We have found, however, that limiting ourselves exclusively to well formatted structures such as database schemas, XML (eXtensible Markup Language) or documents, does *not* simplify the problem in any way except that such documents happen to be smaller, and somewhat more regular. These formatted documents still possess the full array of problems such as incompleteness, irregularity, and inconsistency that our research targets.

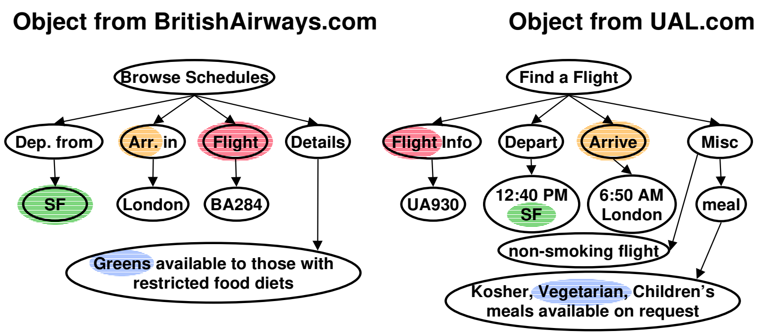


Figure 1.1: Airline Meal Survey

We continue with a detailed illustration of the technical aspects of this dissertation, using a survey of airline meals as a motivating application. In the main body of the dissertation we will show large scale examples and actual results of the use and the reusability of our technology, which are less well suited to this introductory chapter. The data for this example comes directly from the United Airlines [Uni97] and British Airways [Bri97] websites. The particular value of this example is that it clearly exposes the different types of semantic heterogeneity that we deal with in the dissertation. In order to complete the airline meal survey, we need to be able to access information from different airlines' websites. Figure 1.1 shows fragments of objects extracted from the schedule pages of both the British Airways and the United Airlines websites. By inspection, we see that both objects contain information about flights from San Francisco to London, and some details about meals available on those flights. We know that the pages contain the appropriate material to answer human queries about airline meals, and yet the terminology is different enough that we expect an automatic system would have trouble doing so. For instance, we see that the labels 'Flight' in the British Airways object and 'Flight Info' in the United Airlines object refer to the same type of item, i.e., a flight number, but they are in different positions in their respective objects, and they do not provide a justification for assuming they are equivalent. Taking them to be equivalent, we run into our next problem. By extension, we could then assume that the labels 'Arr. in' and 'Arrive' refer to equivalent items. Unfortunately that assumption is only partially true, since one only refers to a location, while the other refers to a time and location.

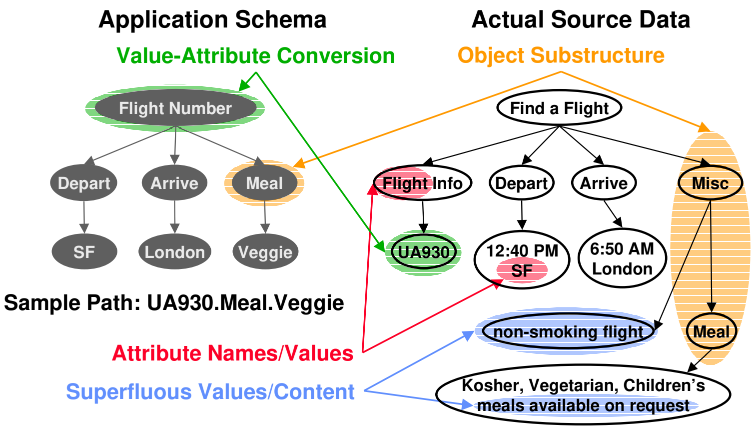


Figure 1.2: Airline Meal Wrapper

Having identified the appropriate material to answer our query (the airline schedule pages) we need to extract the relevant parts (use a wrapper to extract objects from the pages). After we have the relevant parts we then map the objects into a common format to answer queries (write an application specific mediator). Systems exist that enable this work, such as TSIMMIS [PG95]. We go a step further by investigating which parts of the specification process can be automated. In other words, we aim to provide a set of facilities that support the process of identifying, wrapping and mediating the relevant information in heterogeneous data sources. We've seen in Figure 1.1 that different information sources express similar information in different ways. It is typical that the format is also different between information sources and the requirements of any new application which will use the data. Figure 1.2 lists some of the types of mismatch we need to recognize in order to properly wrap the United Airlines schedule page. In particular, we need to know when a value, shown here as a leaf in the structure, is an attribute elsewhere. Attributes in this example are nodes that refer to other nodes, as indicated by arcs between the nodes. For example, the 'Find a Flight' object has an attribute, 'misc', whose value is the string atom 'non-smoking flight' together with a reference to the 'meal' sub-object. Incidentally, United Airlines explicitly indicates that all of its international flights are non-smoking, since they can not assume that all of their potential customers know that they are a smoke free airline. We also must handle superfluous structure as well as content, such as those among the nodes referring to meals. We must also handle variety in the names of attributes and their values.

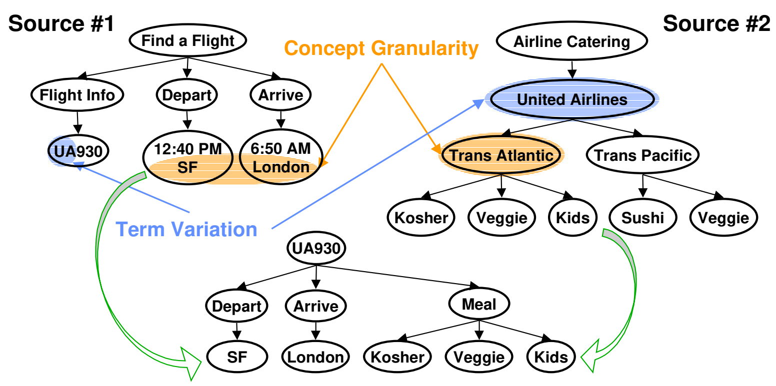


Figure 1.3: Airline Meal Mediator

Finally, when it comes to interoperation of information from multiple sources, we uncover an even wider array of problems to handle. Figure 1.3 shows how the granularity of concepts may be completely different between differing information sources, and that the scope of term variation broadens substantially. In order to properly link the objects for use in our airline meal survey, we must recognize that a flight from San Francisco to London would be categorized as 'Trans Atlantic'. The conceptual granularity differs in that one node referring to an ocean encapsulates some semantics of two nodes representing a start and destination point of a flight. Also, the single 'Trans Atlantic' node subsumes many other pairs of nodes from the other source such as ones for flights between Boston and Paris.

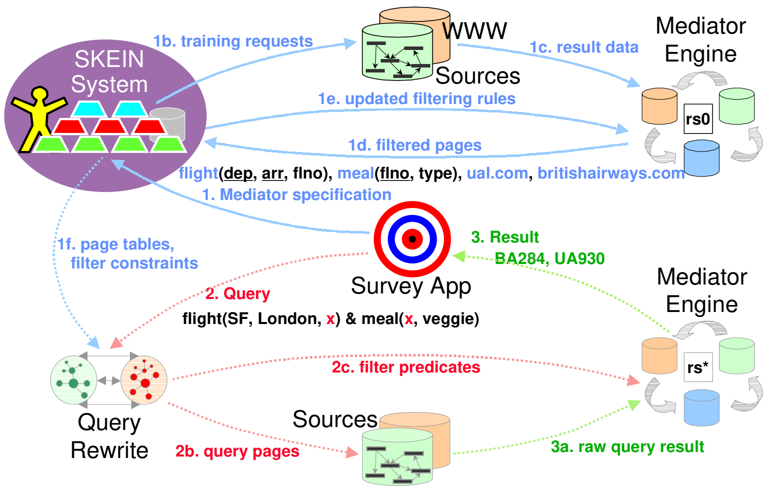


Figure 1.4: The SKEIN System in the Meal Survey Application

We use the SKEIN system to identify in heterogeneous information sources terms which are coherent to those in our application requirements, and recognize terms which are similar across information sources. We also use SKEIN to extract object graphs of these relevant sources for use by mediators that fulfill our application requirements. Figure 1.4 shows a sequence of steps that are involved in using SKEIN to obtain a query result for the meal survey. Starting with the requirements of the airline meal survey application at the center, we have the process of *information refactoring* in the top half, followed by *query reformulation* in the bottom half of the figure.

We assume that the application requires information from one or more sources designed for a different original purpose. Information refactoring is the process by which the relevant source information is cast into a form useful to the application. This iterative process builds a representation of the source information. Query reformulation is a process which optimizes the development of query expressions over this new representation, given the source capabilities. Both processes are necessary to fulfill generic query requirements of such an application. The magnitude of scope of querying semantically heterogeneous information sources is large enough that this dissertation discusses only information refactoring. The slightly lighter shading of the Query Rewrite, Sources and Mediator Engine portions of Figure 1.4 suggest that these features are less emphasized in the dissertation. Since query reformulation relies on the prior refactoring of information in heterogeneous environments, we needed to investigate the refactoring problem first.

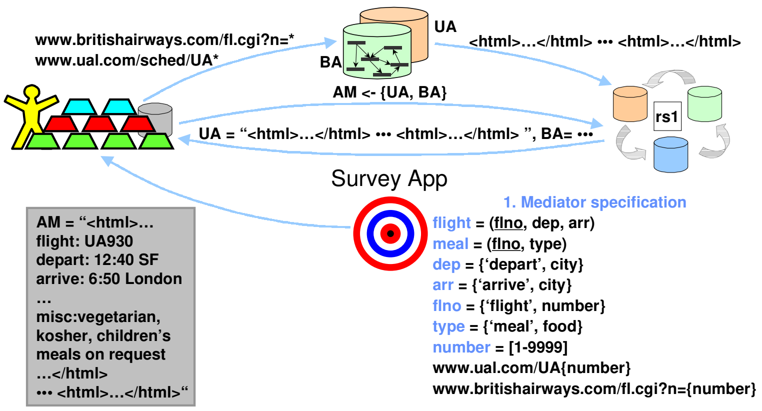


Figure 1.5: SKEIN used for Mediator Specification

Information reformulation for the airline survey application consists of building the mediator to handle the schedule pages of the airlines covered in the survey. Figure 1.5 shows the process of specifying the mediator in more detail. The patterns that describe the survey's requirements are shipped to SKEIN along with a list of web pages that must be crawled to obtain the necessary source data. These patterns form a *mediator specification*, with a list of relations, that can be nested, as shown in the lower right of the figure.

At the start of the process, no information from the sources has been analyzed, so there are no rules yet available to filter the sources. The initial ruleset rs0 in the upper right corner of Figure 1.4 is an empty set, and therefore SKEIN receives unfiltered results back from the information sources. SKEIN then matches up the the specifications to the terms contained in the initial results and generates a modified ruleset rs1 for *expert verification*. After verification, the process can continue, so as to further refine the emerging mediator ruleset. The key point in the process is that it is progressive. SKEIN iteratively establishes matches between the content in the information sources and the mediator specification. Each of these matches builds on the results of the previous matching, and each iteration provides a further opportunity for expert verification of the rules. The rule sets that represent these matches are applied to the source data by a *mediator engine*, a simple rule processor that applies the rules to the source data. SKEIN uses a combination of the position of the information in the source structure as well as the lexical content of the sources to estimate the best matches for each approximation towards the final mediator.

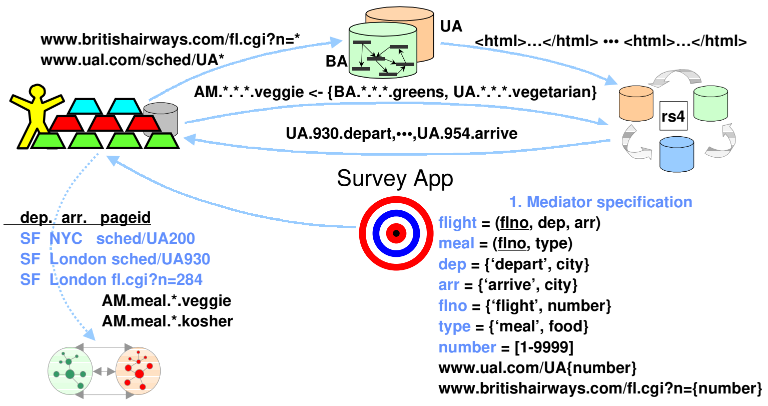


Figure 1.6: Finished Mediator Specification

The ruleset refinement process continues until the execution of the ruleset against the source information returns the desired application data with sufficient accuracy. The termination condition can be determined by the expert using SKEIN, or simply by defining a benchmark test, which the ruleset, now called **rs***, must successfully generate from the source information.

Once the mediator has been fully specified, as depicted in Figure 1.6 it is possible to ask queries against it. For instance, we can now ask about the available meals on flights between San Francisco and London. Since we do not need the full power of SKEIN for query asking, we have a simpler *query reformulation* module, that has no facilities for ruleset refinement, and is fully autonomous. No expert intervention is required to generate query results, since the human effort went in during the mediator specification.

Although the thesis does not cover query reformulation, there is one aspect of it which is relevant to our discussion, depicted in Figure 1.7. The query rewrite module determines how to forward queries to the sources to reduce the amount of superfluous data returned. The rewrite module also sends the filtering rules to the mediator engine, so that the results returned by the sources are transformed into the proper format required by the survey application. The important point is that the query process is symmetrical to the initial specification process. This is not just for convenience, but is crucial when the information sources are volatile. The reason for this symmetry is that when a source changes, the quickest method for updating the mediator rule set is by adapting the existing rules based

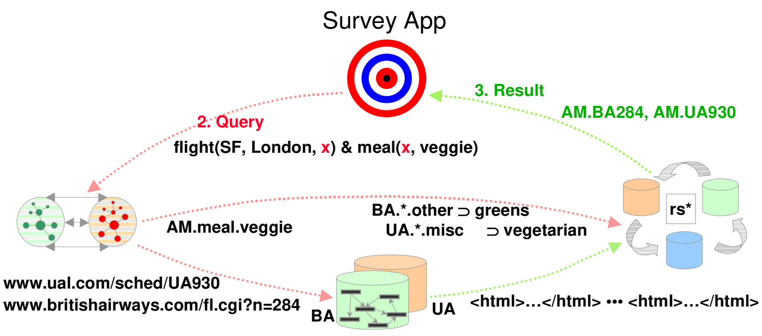


Figure 1.7: Query Answers from a SKEIN Generated Mediator

on the results returned by the mediator engine. The only way to fully leverage this method is by making the mediator generation functionally equivalent to a sequence of successively refined queries.

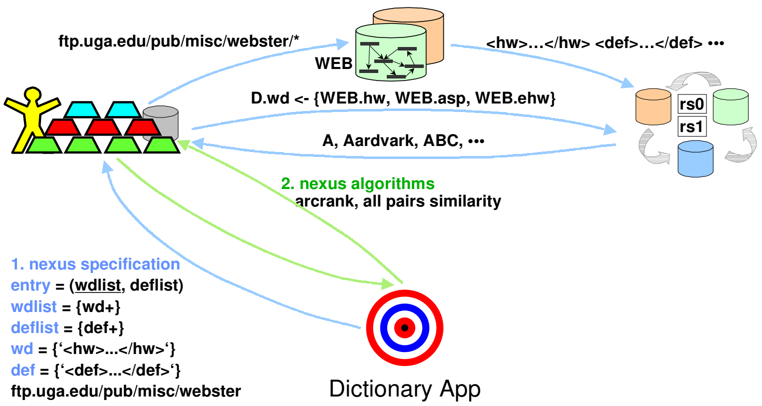


Figure 1.8: SKEIN used for Mediator Specification

We indicated above that SKEIN uses the lexical content of the sources to estimate the best matches between the information source and the mediator specification. The method we developed to enable lexical matches forms the core of the thesis, and is depicted in Figure 1.8. We use a repository constructed from an on-line dictionary to measure the strength of relationships between terms used in the information sources and those in the mediator specification. In order to build this repository, which we call a *nexus*, we need to extract the relevant relationships between words in the dictionary. Since we can not use the nexus to build itself, we do the next best thing: we bootstrap the SKEIN system by using successive approximations of the nexus to refine its own construction. This bootstrapping is the subject of Chapter 3.

To detail the technical requirements of the SKEIN system we recast the information refactoring process into the trapezoid of Figure 1.9. Each layer represents a further refinement of the presentation of the source information to the target application. At the interface of each layer there are issues we must resolve in order to arrive at a correct specification of our research. The SKEIN system incorporates support for each level of the diagram:

1. sources storing information in their individual original formats

2. object structures representing relevant information in each source

3. rules transforming objects and relating them between sources

4. semantic operators governing the presentation of these object structures

5. experts adjusting operators and rules to articulate the sources correctly

6. applications and customers using the articulated source information

Figure 1.9 shows the layering listed above along with the flow of information from sources to the application. In some cases, such as when there exists a pre-existing agreement about the semantics of the data provided by the source, some or all of the intermediate stages are not considered. The information flow will bypass some or all of those intermediate stages between the information source and the application. In particular, when a data source is designed specifically for a given application the intermediate transformations do not need to take place. However, the increasingly common case of heterogeneous information sources and applications requires at least some transformations. In our definition of the process, we use rules to specify the object structures of the sources, and the operators to select the appropriate structures and combine them with others from other sources. The role of the expert is to provide feedback and to adjust the rules as necessary to maintain or improve the quality of the results obtained from the sources.

Often initial semantic transformations are insufficient to cover all of the requirements of an application. We must then resort to multiple iterations of the transformation loop indicated by the *closure* arrows in Figure 1.9. Informally, closure consists of adapting the rules and operators to improve the transformation results. The closure step is repeated until results meet the application requirements. As we've pointed out above in the airline meal

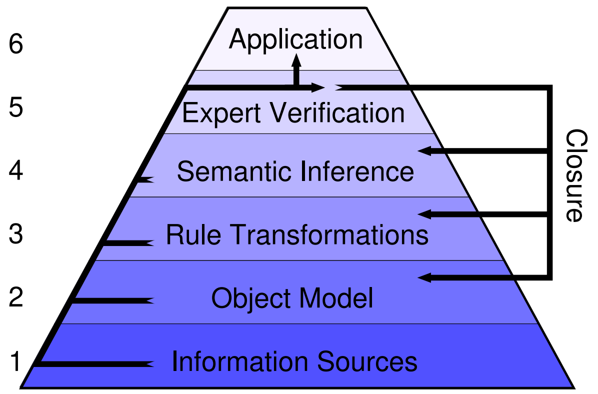


Figure 1.9: Information Refactoring for Novel Applications

survey example this iterated closure is a critical aspect of the problem space. Note that closure does not affect the information sources which are autonomous, and are therefore not available for update by SKEIN.

The next section covers terminology we use throughout the presentation of the dissertation. In Section 1.4 we show how our work addresses the problem statement without making simplifying assumptions. Also, we show our contributions at each of the layers of the problem space. Finally, we show how our motivating example itself has applications within our framework.

1.2 Research Hypotheses

This dissertation focuses on a narrow portion of the information refactorization problem. A single sentence characterization of the question it addresses is:

How do we begin to develop a systematic approach to the interoperation of heterogeneous data sources?

The answer to this question is in three parts, and combines not only information resources but also the framework to create them and the algorithms that use them. In this section we present our hypotheses and the techniques we use to confirm them.

Figure 1.10 depicts the approach taken in the dissertation. Starting from an on-line dictionary, in this case the Oxford English Dictionary, we use a simple rule engine to extract the word nexus. The nexus is a *repository* of all the unique terms of an information source, together with a relationship that expresses how strongly related the terms are to each other.

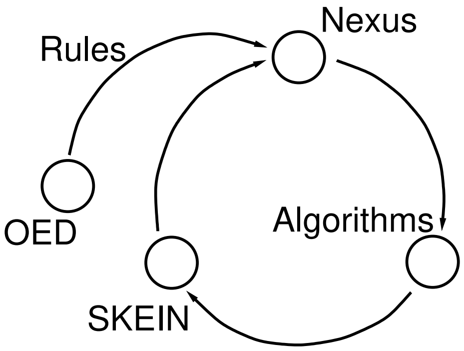


Figure 1.10: Thesis Topics

The nexus is the primary result of the dissertation, as indicated by its prominent position in the figure, but it depends on two other key results without which the nexus is much less valuable. First among these other results are the nexus algorithms which use the nexus to compute the strength of the relationships between terms in the nexus. Without these algorithms it is impossible to determine which terms are most relevant to each other and which terms are most similar to each other. The second of the ancillary results is the SKEIN system which uses the nexus to allow other information sources to interoperate. As implied in Figure 1.10 the three results depend on each other in a cycle, and none is completely separable from any of the others.

1.2.1 Thesis Goals

In this section we present the hypotheses we verify in the dissertation.

Hx. 1 *implicit term relationships contained in dictionary definitions can be efficiently extracted into a nexus that expresses them explicitly*

Hx. 2 *algorithms developed to build and use the nexus scale well and can form the basis of an information composition system*

Hx. 3 *the cost of articulation development, maintenance and reuse decreases when using the information composition system*

Each of these hypotheses corresponds to one of the points in the cycle in Figure 1.10. Each hypothesis has individual merit but is strongest in conjunction with the other two.

Each presents an efficiency, scalability or cost criterion that must be met. We treat each hypothesis in sequence in Chapters 3, 4 and 5. Together the results we obtain form a foundation upon which to achieve systematic semantic interoperation. In the next sections we look at the issues brought up by the hypotheses.

1.2.2 Efficient Nexus Construction

This section concerns the efficiency of the construction of the word nexus. The work presented in Chapter 3 illustrates the development of an articulation over an on-line Webster's dictionary source. This articulation defines the transformation which builds the word nexus we've described above.

1.2.3 Nexus Algorithms Are Scalable

This section concerns the scalability of the algorithms developed using the word nexus. The dictionary repository explicitly models two relationships present in the source data. First, term x appears in term y's definition, and second, term x uses term y in its definition. What is less clear is that the structure of the repository allows for the computation of semantic relationship between words. For instance, we can roughly compute the similarity of terms based on the number of words they share in their definition. Algorithms over the graph defined by the structure of the nexus and described in Chapter 4 expose these implicit relationships for use in other applications. These extensions to the SKEIN system are crucial for the development of the word nexus.

First, we compute structurally significant definitional terms. These are the terms that are frequently used in definitions. As expected, articles and common prepositions dominate this ranking. On its own this ranking does not help to compute the relative importance of definition words to the words they define. Indeed, articles and prepositions are typically considered *stop words*, of little semantic importance in a word's definition. The insight we obtain from this ranking is that few words contribute significantly to the words' rankings. The measure we finally selected to compute relative word importance is one that relates the contribution of one word's rank to the ranks of the words in its definition.

Second, we can go on to use the relative importance measure to find similar words in separate structures, based on the similarity of their usage rankings in the dictionary repository. For example, in Figure 2.1 on page 39, the binary articulation between the

OPEC web pages and the world factbook needs an initial assessment of the most likely common vocabulary between the two sources. The nexus and a word similarity computation provide this functionality.

To summarize, we use semantic relationships between the graphs terms to support the development of other articulations, and we compute these relationships from the repository's structure. These algorithms are:

1. Arc Importance which computes significant definitional terms, (Section 4.1.2)

2. ArcRank which determines relevant arcs between terms, (Section 4.2.1)

3. All Pairs Similarity which finds classes of similar objects. (Section 4.4.3)

1.2.4 SKEIN Lowers Articulation Costs

This section concerns the information refactoring phase of mediator development that is the heart of the SKEIN system. Recall that in this dissertation we consider varied information sources, from plain text to relational tables. We assume that, in a first step, source information is represented by proxy objects generated using AMORL, the rule language for the AMO object model defined in Section 2.1.4. We will see that the proxy representations and the rules that generate them are iteratively developed in the course of applying operators to the source. The notion of approximate solutions that are iteratively improved is central to all of the operators presented in the following subsections. The iterative process is also followed when considering an articulation over two pre-existing ontologies for a new target application. This similarity suggests that it is possible to consider the problem of articulation on well defined source ontologies, while also examining the development of ontologies, via the articulation mechanism, from arbitrary data sources such as plain text or XML tagged data.

Articulation Development

The presentation of Chapter 3 introduces the operational framework that performs the necessary transformations to the dictionary for a target application. The target application of this articulation is the computation of semantic relationships between the dictionary's defined terms, based on the words in their definitions. We have generalized the definition of articulation in Section 2.1.2 so that it covers refactoring of a single source with respect to a target application. Thus, an articulation is a sequence of operators, such as those listed below. The result of applying an articulation to the sources it transforms is a new ontology, which may or may not be materialized. Each operator applies an AMORL ruleset to objects of one or two sources, and returns the result of its operation to the target application. Articulations are unary or binary, based on the number of sources they operate on. Articulations initially operate on a rough model of the source or sources, and are refined iteratively to meet the requirements of the target application. The sequence of operations listed below follows the order in which the raw dictionary data is transformed into a directed graph with head words as terms that label the nodes of the graph.

Glossarize

To construct a repository from the dictionary we enumerate all of the items that must be considered to be individual, and then generate proxy objects for them.

Extract

To associate definition terms with each entry we must find those terms within the dictionary entry, and enter them in a list associated with a proxy in the sequence from above.

Match

To link dictionary entries using definition terms we need to match each extracted term to a proxy in the glossary, and generate a link to this proxy from the proxy it associated with in the previous step.

These three operations illustrate both unary and binary operations from the algebra. The unary operations demonstrate the preparation of the relevant portions of the dictionary source data. The use of a binary operation on information from a single source shows that single sources can contain inconsistencies when refactored for a new application.

Articulation Refinement and Maintenance

The nexus extracted from the Webster's dictionary [MIC96] and described in Chapter 4 contains just under 97,000 defined objects, referred to by two million arcs, and requires just under two hours to generate from the source data downloaded from the ftp (file transfer protocol) site [MIC96]. A significant aspect of the development of the articulation is that it did not initially produce a high quality repository. However, the initial results were good enough to bootstrap improvements to the articulation. Indeed, we analyzed the data that did not contribute to the repository, i.e., the exceptions in the data, to generate new rules that improve the articulation. In addition, the dictionary serves as its own spell checker. One type of exceptions occur when the defined term in the dictionary is misspelled. The iterated refinement of the dictionary shows how a knowledge source can bootstrap its own development.

The maintenance problem for the dictionary repository is the following: the source maintainer's goal is to add updates and revisions to the dictionary, while also correcting errors discovered in the original data. At irregular intervals, approximately semi-annually, these changes are published to the ftp site. The cost of maintenance is the effort it requires to adapt the articulation and regenerate the repository to the level of consistency it had prior to the update. We measure the cost of adaptation in two ways. First, the number of rules added and deleted in the articulation's operations. Second, the amount of time spent between the time the new source data is available, and the time a revised repository, based on the new data, becomes available.

Articulation Reuse and Scalability

The ability to reuse the articulation on a different source for the same application demonstrates that articulations can also save development time. We performed two experiments, one to for reuse, the other for scalability. In the first, we constructed a small scale repository based on Roget's thesaurus [PRO99b], with one thousand nodes and 5,100 arcs between them. To confirm the quality of its structure we computed an *all pairs similarity* algorithm over the entire repository. This algorithm finds all the pairs of nodes which have arcs to and from the same sets of nodes.

The second experiment confirmed the scalability of the Webster's dictionary articulation. This experiment required modifying the articulation to accept the Oxford English Dictionary [Oxf99] as a source. At around ten times the size of its original source, the 570 MB raw data file converts into a 327,000 object, eight million arc graph. When assessing the effectiveness of articulation scalability, we consider the time required to modify the articulation, the time to compute the graph, and test the data on a real problem, together with the original Webster's repository.

In this section we have asked what are the enabling steps towards interoperation of heterogeneous information systems. Our response implies that we need a combination of an operational framework to perform necessary transformations, and a vocabulary bootstrapped from the framework to assist in subsequent transformations.

1.3 Terminology

The research in this dissertation is at the crossroads of database systems and artificial intelligence. Since this work considers the problem of composing information from multiple heterogeneous sources, it is fitting that the first task of importance is to reconcile the vocabularies of these fields in the context of this work. Indeed, the terms relating to the problems covered in this dissertation are not the same in the two fields, nor are the definitions of these terms entirely consistent within the same field.

We will generally use the terminology listed below in boldface, and it is preferred over the other terms listed directly below their definition. In Chapter 2 we provide formal definitions for the most significant terms above as they apply to the thesis. For now, to clarify the discussion that follows, we provide informal definitions for the terms below:

Object

abstraction of a physical entity, or a concept, e.g., meal

- Tuple/Instance/Frame

Term

named reference to an object, i.e., flight.misc.meal

- Reference/Slot/Attribute

Domain

information scope where consistent use of terms is implicit; UAL Web schedule

- Class/Relation/Frame model

Relationship

mathematical relation between objects of a domain; meal $\sim$ vegetarian

- Facet

Context

object that specifies semantic consistency constraints on sources

- View/Microtheory

Wrapper

explicit specification of information in an underlying source

- Extractor

Articulation

explicit linkages between multiple sources and a target application

- Ruleset/Mapping/Mediator/Facilitator

Repository

information source associated with a domain, i.e., www.ual.com

- Source/Knowledge Base

Ontology

set of terms and the relationships between them; meal, style, kosher

- Schema/Metadata/Model

We present an example centered around the term 'dog.' The dictionary does not associate a gender with the generic definition of dog. Dog in this context is associated with both genders of the animal. However, in the specialized domain of dog clubs, the object 'dog' is only an abstraction of the male of the species. An ontology which relies on both repositories for its definition of dog must reconcile the differing views with an articulation. To convert the representation of dog in one representation to that of the other, we might use a rule such as the following:

`x` $\in$ `dictionary & x.species == dog & x.gender == male` $\Leftrightarrow$ `y` $\in$ `kennelclub & y.gender == dog`.

Note that 'dog' appears both as a value and as a schema element in the examples above. This situation occurs frequently in general information sources. Henceforth we hold to the definitions given in the vocabulary list above. We will provide formal definitions as necessary in the body of the dissertation. The only cases in which the above vocabulary will be used with differing definitions is to describe related work when that work's definitions deviate from the above. We now define some terms which are derived from the results of our research, but have not generally been used in either the database or the AI community.

Interoperation

use of autonomous sources without affecting their autonomy

Consistency

relationship specifying that given term meanings be everywhere equivalent

Coherence

relationship of relevancy of source data to requirements of a target application

Similarity

relationship of semantic relatedness between information in distinct sources

Closure

constant outputs between iterations of an updated articulation over a source

1.4 The SKEIN System and Word Nexus

This research introduces a novel approach to describing the process of extracting information from disparate sources and enabling it to interoperate. Figure 1.11 shows how the results of the work would be applied in a practical setting such as supply chain management.

There are three sections in the figure, each representing an information interoperation activity. The rightmost section depicts a supply management problem which is solved by the use of an articulation between a supplier database and a facility database. Before creating such an articulation we need to have an articulation development toolkit, shown in the central section of the figure. The articulation toolkit is supported by an articulation of databases, free text and structured documents geared towards industrial applications such as supply chain management. The domain repository associated with this toolkit is instantiated from a general purpose articulation generator in the left section of the figure. This system is developed by bootstrapping an articulation that constructs the OED nexus. Note that all of the transitions between sections of the figure are supported by the verification of an expert, and that a feedback loop allows for improvement of the articulations. Our research contribution, illustrated to the left of the dashed line, is the nexus and other tools that form the core of the SKEIN system. SKEIN in turn simplifies the job of constructing articulations between diverse information sources, such as supplier and facility databases, as shown in Figure 1.11.

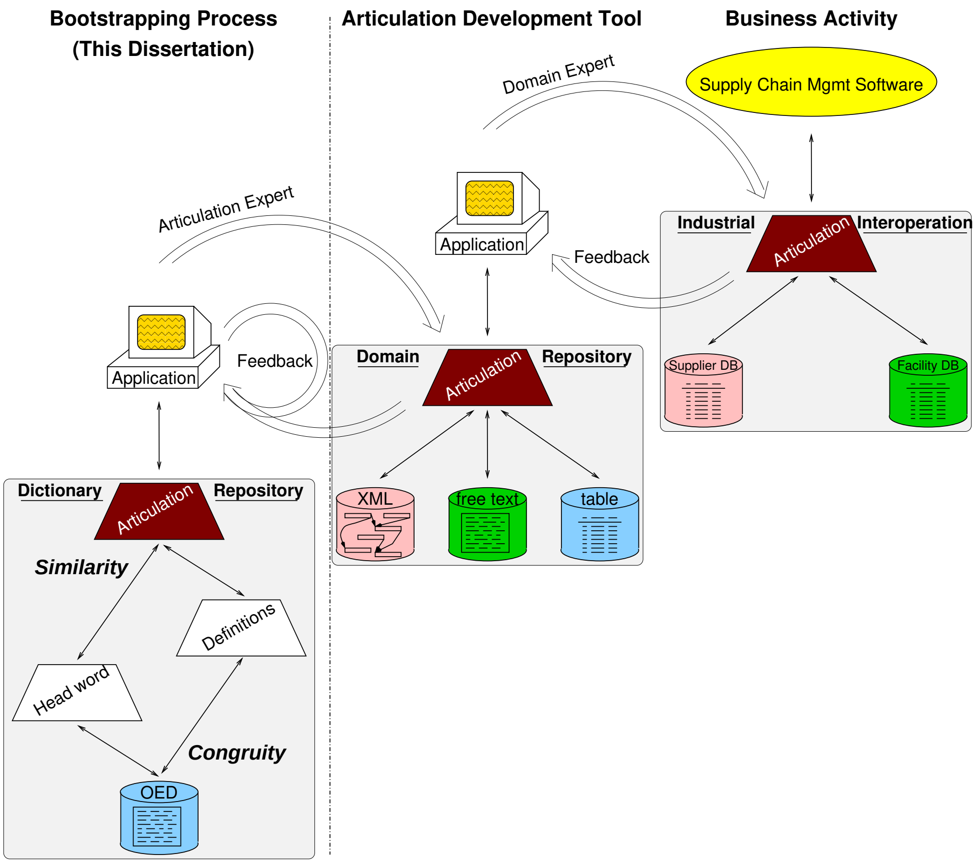


Figure 1.11: Interoperation in Supply Chain Environment

The following three contributions, which we first introduced in Section 1.2, form the core of this dissertation:

1. dictionary repository of semantic relationships among terms (**Nexus**)
2. algorithms for extracting structural relationships from graphs (**extensions**)
3. framework for rapid development, and maintenance of articulations (**SKEIN**)

These contributions are made possible by the development of the simple framework of an object model and rule language. These two form the basis of an ontology algebra, which allows an algebraic expression of information extraction and composition. The algebra, important in its own right, is incompletely specified in this dissertation, and is presented in Chapter 5 simply as infrastructure to the main body of work.

As a preliminary step in building the SKEIN system we developed a simple object model, AMO (Atoms Modeled in Objects), in the spirit of OEM [GCCM96] and YAT [CDSS98]. AMO represents objects, their contents and the objects' relationships to other objects. AMO is designed to maximize simplicity of conversion from HTML, XML, database tables, as well as text files. The model captures both the reification of uninterpreted data into objects, as well as the transformation of objects to relate them to others. We use AMO as our bridge between the first and second layers of the problem space as depicted in Figure 1.9.

We construct and maintain repositories with the rule language AMORL. Repositories are collections of related objects whose relationships can be represented as a directed labeled graph. AMORL consists of nine functions on object patterns. These nine functions are classified into the categories of constructors, connectors, editors and converters. An object structure can be transformed into any other using these rules, and the common portions of any structures can be mapped into a single structure. Object patterns are expressions that identify zero or more objects in a repository. AMORL bridges the representation gaps between object structures extracted from different sources.

In order to demonstrate the scalability and efficiency of our work through an example, we use an on-line dictionary. Initially we worked with a Webster's dictionary, and then scaled our efforts to a much larger source, namely the Oxford English Dictionary (OED), second edition. We begin by extracting a glossary of terms from the head-words of the dictionary, and separately, a set of definitions that match the glossary. Then we generate a graph from the glossary, such that each use of a word in a definition results in an arc between the defined and defining word. The extraction of a graph requires knowledge of the dictionary domain, as well as the semantics of the language used to express the definitions. The combination of the glossary and definitions into a graph requires knowledge of English morphology and the idiosyncrasies of usage in the dictionary. Manipulating the OED data set is not a toy problem. Its 511,000 defined terms, including approximately 180,000 exact synonyms, and eight million definition words defy any attempts at systematic manual treatment.

Having generated the dictionary graph we demonstrate our application of the graph structure. We apply a ranking algorithm to the nodes of the graph, then to its arcs, in order to express for each given term which are the most important terms in its definition and which are the most important terms that use the given term in their own definition. With this ranking, we are able firstly to generate classes of terms that have strong kinship to each other and secondly to determine which terms subsume others, and which terms they specialize. This is a demonstration of a semantic decomposition of a large graph structure into smaller hierarchies. The benefit of this decomposition is that these classes of related terms may be used to bootstrap the process of relating other information sources that use varying nomenclature.

A second algorithm we introduce is one which computes the similarity of all pairs of objects in a directed graph, based on the similarity of the objects with which they share arcs. We developed this algorithm on the Roget's thesaurus graph, which we built to test the reusability of the Webster's dictionary articulation. Just as the arc ranking algorithm allows us to measure the coherence of objects in a graph, the similarity algorithm estimates the similarity relationship of objects in a graph. These two relationships in turn are the foundation of our semantic algebra.

The semantic algebra is a higher level language built on top of the rule language, in the same way relational operators depend on selection conditions, join parameters and projection attributes. The algebra is a set of composable operators that transform object repositories into new object repositories. The novel aspect of the algebra is that a form of closure is applied to the operators, such that each successive application of the operator to the initial source represents a change to the exported object graph. The iterative application of the operator terminates when a fixpoint is reached (and the repository no longer changes), or when an alternative termination criterion has been met.

Our presentation of the semantics of the algebra shows that it adequately represents the creation, refinement and maintenance of articulations over information sources. In particular, we discuss the problem of finding matching terms between pages of the British and Finnish government websites, as linked to by the NATO website. We show how the use of the OED nexus to support the `match` operator of the algebra obtains 70% of the matchings of terms that a human expert determined were correct. Also, the match operator did not produce any so called false positives or inappropriate matches. These encouraging first results indicate that the nexus is a crucial part of the SKEIN system.

There has been a significant amount of scientific investigation touching on various aspects of this research, but semantic heterogeneity has also been a subject of intense study in philosophy, literature and art. Appendix C devotes space to the the non-scientific exploration of heterogeneity, while the next section covers the conventional research literature.

1.5 Related Work

The starting point for the work in this dissertation comes from a proposal for an algebra for ontology composition [Wie94]. This proposal foresees problems in maintaining knowledge combined from autonomous knowledge sources. In this section we examine some of the other work that is relevant to the thesis.

1.5.1 Research in Integration Fundamentals

Integration of data, knowledge and systems is an area that has been gaining attention in recent years. The relevance of interoperation as opposed to integration has become manifest with the growth of the Internet. The literature on object models and rule languages appears in a wide range of domains, so we restrict ourselves here to research in the database field. Citations on formal contexts and articulations appear primarily in the area of artificial intelligence.

Most recently, technology transfer to Gigabeat Inc. [Gig00] allowed a generalization of the ArcRank algorithm of Chapter 4 to measure the similarity between musical artists, as well as the similarity between songs. A new visualization technique, the affinity chart, also described in Chapter 4, was invented to simplify the comprehension of and the navigation through these large data sets. Model management [BLP00] is a new field that aims to achieve exactly the same goals as the work in this dissertation. One of the stated goals of model management is to achieve the automatic specification of transformations of mediators to support new or changing data and changing application requirements. The lack of a method for iterated improvement of mediators, and the lack of a structure such as the nexus do not diminish in any way the importance of this work.

Semantic integration is the focus of the research leading up to the Infosleuth agent architecture [BJBB+97], in which ontology agents mediate in cases of semantic heterogeneity. The derivation of the domain models used by the agents is not considered. However, the establishment and maintenance of these ontologies has a high cost and should not be ignored. The Infomaster system [GKD97] establishes a reference schema for the information it integrates. In practice each application requires its own schema, so a double translation is necessary for any information the client applications receive from data sources. As the reference schema must serve many applications, it must also be extremely general, and simultaneously highly detailed. Such a schema is sensitive to any change in the sources it integrates. The difficulties in creating and maintaining a reference schema independent of client applications are not considered in Infomaster.

We have anecdotal evidence from the European Esprit program about the cost of producing such a reference schema. One of the Esprit program's goals was to to provide a fully object oriented schema to enable the integration of automobile manufacturing information between all of the German manufacturers. After years of specification efforts the schema remained incomplete, and too general to allow any individual manufacturer to rely solely on it.

The TSIMMIS project [PG95] introduced the object exchange model (OEM) [GCCM96], which predates the extensible mark-up language XML, and is quite similar to it. The IFO model [AH87] considers semantics, but does not focus on issues relating to heterogeneity. YAT [CDSS98] is an object model that has the property that it is self describing, and can represent heterogeneous sources, but lacks facilities to restructure them for consistency.

Sequence Datalog [MB95] is a language that can manipulate ordered sequences. The ability to handle sequences of objects is a vital part of restructuring languages. The reason sequences are so vital is that they can represent the ordering of objects within a document. Mathematical sets and relations are not suited to this document management task. The rise in importance of HTML and XML accentuate the need for tools that support sequence processing efficiently. Editor [AM97] is a computationally complete language for restructuring text documents, but does not have an object model. This language has a well-defined class of programs that are polynomial-time restructurings, and therefore efficient.

Description Logics (DL) [Hul97] are used to represent the information content of sources. There is also discussion of efforts to integrate DL with query languages. Semistructured data has become a significant area of database research. When such data is updated, its structure frequently changes shape in complex ways. Tracking change in semistructured data makes it possible to efficiently reconstruct previous versions of the data, by applying edit scripts to the current structure. The C^3 project [CG97] introduces a set of primitive operators on objects in a semistructured data graph, in order to express change in the graph as an edit script. This work does not consider semantic inconsistency, as the graphs are assumed to be snapshots in time of the same structure.

1.5.2 Research in Algebraic Methods

The notion of articulation between *microtheories* [Guh91] originated in the AI community. Microtheories are contexts that are statically linked through their articulations into a larger complete ontology, such as the CYC system. The CYC system is a handcrafted ontology that has been applied for years in various technology demonstrations. It has not supported transparent transformation of microtheories, and the repercussion of any transformations on the articulations are not well understood. This system also ignores the need for maintenance when knowledge about a domain changes. The Open Directory [Net99b] is a novel approach to the construction of a large information repository, which relies on thousands of editors to maintain the structure as it grows. The editors serve in effect as articulation agents for their areas of expertise. Multiple definitions are common, and conflicting or contradictory statements by different editors are accepted. All maintenance in these systems is fully manual.

There exists some work on the use of context [KS96] to handle database object semantics. This research uses a semantic proximity relationship, but does not discuss relevance of source data to a target application. The paramount importance of the targeted application in our research can not be overemphasized. We contend that every information source is tuned to its original application, and that any interoperation of information must explicitly express the conversions that allow sources to be used in a new setting. Gradations of a consistency relationship [HM93] can be used to relate information sources, when consistency also encompasses relevancy. In considering semantic reconciliation between data source and data receiver [SM91], some issues relating to coherence are raised. That work does not discuss maintenance issues nor the actual algorithms used to achieve semantic reconciliation.

TSIMMIS is geared towards query answering, and wrapper specification is considered orthogonally to query answering with a separate mediator specification language (MSL). A recent refinement of this language, TSL [VP99], or tree specification language, has more efficiently computable capabilities.

TSIMMIS enables the building of wrappers and mediators that are capable of handling some semantic heterogeneity, but not systematically nor explicitly. For example, a new relation may be created by manually defining the join of two relations in underlying sources with semantically differing names. Wrappers may be coded to resolve conflicts and inconsistency, but do so silently. Such implicit semantic changes are difficult to maintain as the underlying source changes.

Query languages for semistructured data such as Lorel [AGM+97], UnQL [BDHS96], XML-QL [DFF+98] skirt the issue of heterogeneity entirely. Wrappers that provide data for these query languages are generated independently, specified with a separate language, and must be maintained separately from the query system.

Recent theoretical results define the problem of finding common schema in semistructured data [NAM98], which roughly corresponds to our `Intersection` operator. MDM, the data model of the Rufus system [RS91] describes *Aggregate* and *Partition* operators, which we subsume in the `Summarize` operator we will introduce in Chapter 2.

The Onion system [MKW00], building on the work presented here, investigates the problems of applying the `Intersect` operator to semistructured data. A taxonomy of information extraction techniques [CJN+00] provides a foundation for the choice of the `Extract` and `Filter` operators.

1.5.3 Research in Repository Integration

In the past few years the problems associated with ontology merging and alignment have become active areas of research. The medical informatics field has played a pioneering role in integrating information from multiple varying sources.

The medical informatics community has had a long involvement in efforts to integrate and reuse separately designed and maintained knowledge sources. The UMLS system [COS98] is the largest of these systems. Recent efforts to improve access to these integrated knowledge sources include [Pra97] and [OSSM99]. Semi-automatic integration of medical thesauri [DHR+98] could greatly simplify the interaction between insurance, pharmaceutical and medical information for medical patients. Research in integrating thesauri for automatic translation [NTR98] is ongoing, and is highly valuable in governmental and diplomatic systems. Governmental organizations are also involved in efforts to enable mediation between autonomously developed systems, as evidenced by work on environmental restoration of INEEL [PHW+99].

A rule-based approach to semantic integration [BCV99] uses a description logic to generate a shared ontology for the source information, and rules to map terms to the common format. However, the notion of maintenance and considerations of scalability are lacking from the discussion. Also, there is no description of the integration process, as an application of an algebra over the source domains. The operations performed in Section 3.2.2 bear a resemblance to the alignment operation [CHR97] used to reconcile some of the terms of the CYC upper ontology of 5,000 concepts with the Microkosmos ontology of roughly the same size.

The fundamental role of placing semantic relationships into an algebra is unique to our work, but the use of relationships themselves appear in other work as well as ours.

WHIRL [Coh98] uses textual similarity to find co-referent terms in distinct sources with high accuracy. The textual similarity measure it defines is one example of the multitude of initial similarity measures that approximate the similarity relationship between sources.

Techniques for the transformation of ontologies and program specifications are discussed in the context of category theory. While this work describes transformations where the domains are complete or exception free, it lacks flexibility to deal with real world data and erroneous specifications.

The problem of reuse of ontologies in new environments is similar to the problem of articulating information for new applications. An ontology of numerical properties is retargeted for use in an engineering project at Boeing [UHW+98], and a handcrafted adaptation of the original ontology is presented. This work applies a one-time transformation of the ontology to fit the requirements of the target application. The notion of maintenance is not considered in this work, as changes to the specific source ontology are infrequent.

Early work on specification morphisms [Smi93] develops a system to iteratively refine a program specification into a working application. Our work follows similar principles for deriving contexts from sources, and refining them for use in a target application. We extend the notion of specification to allow for inaccuracy, inconsistency and incompleteness found in real world information sources.

1.5.4 Research in Repository Algorithms

Dictionaries have been the subject of computer investigations for approximately forty years. Large scale lexical corpora have been constructed in the the past ten years, and automated construction of these thesauri has occurred in the last five years. In particular, the computational linguistics community is interested in so-called semantic parsing of dictionary entries.

Some early work on constructing taxonomies[Ams80] and extracting *semantic primitives* [Dai86] used a graph generated from the dictionary definitions. MindNet [RDV98] is an example of a *lexical knowledge base* that relates terms according to some two dozen relationships. MindNet is generated by phrase parsing an online dictionary. This system attempts to distinguish semantic relationships between terms by the sentence structure and keywords in the definitions. A different and freely available system based on the Webster's dictionary data is in early development [Res98]. Such systems have limitations relating to the sparseness of the structures that are extractable from dictionaries, and the inaccuracy and slowness of phrase parsing techniques.

The WordNet system [MBF$^+$90] is a corpus of nouns, verbs and adjectives that lists lexical relationships between entries in the system. WordNet was manually constructed by a group of linguists. It is publicly available and has been widely used in other linguistic and computational research. It is specific about the relationships between entries, but is therefore limited to a small set of possible relationships. Also, it separates the four major parts of speech (noun, adjective, verb, adverb) into separate categories, and is comprehensive only to the extent that the developers were interested in particular topics

A number of new algorithms exploit the structure of large graphs to extract semantic features from them. The main focus of these algorithms is the World Wide Web, but these algorithms apply to directed labeled graphs in general.

The PageRank algorithm [PB98] is used in the Google search engine [Goo99] for ranking the importance of web pages. PageRank is a flow algorithm over the graph structure of the World Wide Web that models the links followed during a random browse through the Web. Google's phenomenal growth over the past few years from a two person research project to a system running on several thousand servers supporting over fifty million queries every day is entirely due to the success of PageRank. This algorithm is the starting point for the ArcRank algorithm we use to determine the strength of the relationship between dictionary terms.

Latent semantic indexing (LSI) [DDL$^+$90] and hypertext *hubs and authorities* [Kle98] exploit properties of eigenvectors to answer queries over a corpus of text documents or web pages. The eigenvectors are computed from the adjacency matrix of a graph representing the structure of the corpus. These methods reject, a priori, stop words such as 'the' or 'and,' and any words that appear overly frequently in the document corpus. LSI also fails to measure the relative importance of relationships between terms. However, the underlying mathematics of these systems are close to those of our dictionary nexus.

In the domain of the World Wide Web, there is a great interest in finding related pages. Most Internet portals, such as Netscape [Net99a], and news services now provide access to similar pages or related articles in their collections. Dean and Henzinger have done research on finding structurally similar pages [DH99]. An efficient all pairs proximity algorithm [GSVG98] finds which objects in a graph are most strongly linked.

The relationship extraction algorithm we present in Chapter 4 extends and improves techniques pioneered in the DIPRE algorithm [Bri98]. There is some other work that uses the notion of patterns and sample sets to extract relations [GW99]. These algorithms begin with a sample set of data that follow a fixed structural or lexical pattern. The first goal is to find in a large data set other instances of the data items. Then any new patterns that contain the data items are identified. Now it becomes possible to search for any data instances in the large data set that conform to the new patterns. The above procedure can be iteratively applied until a fixpoint is reached and neither new data nor new patterns are uncovered. There is also a research direction that seeks to uncover universal techniques for extracting relations from web pages [CF99]. This work is based on approximate matching techniques. The notion of related web pages in [DH99], and that of clustering search results [ZE99] are closely tied to the kinship relationship we extract from the dictionary data. In a similar vein to our work the CLEVER group at IBM [pro99a] uses the hubs and authorities approach, as defined by Kleinberg, to cluster web pages.

1.6 Thesis Outline

The remainder of the dissertation proceeds as follows. Chapter 2 presents the groundwork for the following chapters. This work initially appeared in [JPVW98], covers a simple object model and rule language, and presents some key definitions. Chapter 3 describes the Webster's dictionary source and the iterative process of defining a wrapper over the source. This work was presented in [JW99] to demonstrate the maintainability of such wrappers when the underlying sources change. Chapter 3 then shows how the work on the Webster's source was extended to the much larger Oxford English Dictionary, with little additional effort. Chapter 4 presents the ranking algorithms which enable restructuring of newly incorporated sources. [Jan99a] describes these algorithms. Chapter 4 also describes iterative techniques for extracting subsuming, specializing, and kinship relationships from ranked sources. To provide further infrastructural details, Chapter 5 investigates properties of an algebra defined over wrapped sources. This is work that appeared in [JMN+99]. Conclusions and directions for future work are given in Chapter 6. Appendix A describes the relationship between our object model, and semi-structured data, such as XML. The appendix shows how to convert between XML and our model, and provides XML DTDs for the nexus, and the visualization scheme we use to navigate the structure of the nexus. Appendix B presents scripts which transform the raw source data from the OED into two relations, the first associating the head words to a numerical key, the second all definition words to the key of the head word they define. The scripts then generate the nexus from these two relations, and finally use the nexus to match NATO government websites. Finally, Appendix C provides a few examples of the extreme cases of semantic heterogeneity present in art and literature. These cases provide amusing anecdotal evidence that fully automated integration of information sources is not feasible.

Chapter 2

Groundwork for an Algebraic Approach

Chapter Outline

In this chapter we provide an overview of the background for the thesis work. We begin by defining ontologies, domains and articulations. We proceed with a simple object model, AMO. We present a definition of context that we use to encapsulate object graphs. Contexts are the unit of semantic consistency in our framework and are the operands of the algebra. We continue with a presentation of AMORL, a language of primitive rule operations that transform objects. We begin the presentation of AMORL by listing the algebraic operators, in order to motivate the sections that follow. We then define and motivate our model of semantic consistency, considering two inherently semantic relationships: coherence and similarity. While the two relationships are very close, distinguishing the two is important for scalability. We begin with the foundations of the SKEIN system.

2.1 Architecture

The amount of potentially useful data available on the Internet is growing at a tremendous rate. There are thousands of sites which contain overlapping or complementary information. It would be most convenient to access related information in a uniform fashion through a single interface. For example, there is enough freely available data on airline schedules and fares, car rentals and hotel reservations to plan any trip. However, without a unified access interface, in which all of the data is presented uniformly, it remains much easier to book travel arrangements through an actual travel agent. In many domains such as the sciences, freight shipping, construction and pharmaceuticals, such professional intermediaries do not exist at all. While some programmatic interfaces do exist, they are handcrafted and require specialized tools to maintain.

To remedy this state of affairs we consider two fundamental obstacles to the development of such interfaces. First, we examine the task of identifying and keeping track of exactly the relevant portions of each data source, a task which is complicated by the *coherence problem*. Coherence was defined in Section 1.3 as the relevance of information from a source to the requirements of a target application. The coherence problem, in brief, is that there is no automatic procedure to determine what elements of an information source are important for a given application. As a substitute, we define a coherence score to *estimate* the true coherence $C_{x,t}$ of a term x to a target application t:

$$0 \leq C_{x,t} < 1 \tag{2.1}$$

The coherence problem, discussed in more detail in Section 5.2.1, occurs because sources contain not only superfluous material, but also incomplete or partially incorrect data. Note that we use superfluity, incompleteness and incorrectness not as absolute terms, but only relative to a new application for which the sources were not originally designed. In order to compute the coherence score we need to come up with a procedure to approximate the true coherence relationship.

Second, we describe the problem of identifying the parts of different sources that overlap, because they are identical or have related content. We call the recognition of this overlap the *similarity problem*. In the introduction we defined similarity as the relatedness of information in distinct sources. Again, the similarity problem refers to the lack of an automated procedure to compute similarity. As for coherence above, we define a similarity score $S_{x,y,t}$ which stands in for the exact similarity relationship:

$$0 \leq S_{x,y,t} < 1 \tag{2.2}$$

The similarity score estimates the relatedness of two terms x and y, in the context of a target application t. The similarity problem is treated in Section 5.2.2, and it arises out of the autonomy and semantic heterogeneity of distinct information sources. The coherence and similarity relationships are inherently semantic and are in general not automatically reducible to a program or mathematical expression. We approach this limitation by considering how to compute approximations of these relationships. Therefore we introduce the notion of coherence and similarity *measures*. Coherence and similarity measures are explicit mathematical *expressions*, written in the form of a sequence of rules, that estimate the coherence and similarity relations defined above. Such *scripts* of transformation rules are created, refined and maintained interactively with a human specialist, using an algebra to expose individual pieces of the relationships.

The ultimate goal of the SKEIN system is to implement the operators of an algebra. While the complete algebra is beyond the scope of the dissertation, we have defined a few operators. In order to define an algebra we must first chose and define the operands of the algebra. The algebra operates on semistructured data, as represented by directed labeled graphs of objects described in Section 2.1.3. Inputs to the operators, as well as their outputs, are instances of these directed labeled graphs. These graphs are assumed to be connected components. We define these connected directed labeled graphs below as our unit of semantic consistency. The nodes and arcs of directed graphs model the terms and relationships of these data. We require connectedness because we consider disjoint graphs to have no a priori semantic relationships; they are therefore independent information sources, or *contexts*. Contexts in our framework are a representation or mapping m of the objects in a directed labeled graph, to items in the real world. In our system disjoint graphs may *articulate*, that is, join together, according to a similarity measure defined by application requirements.

Each algebraic operator listed in Section 2.1.5 takes as input graphs of semistructured objects and transforms them according to a coherence measure or a similarity measure. Having directed graphs both as inputs and outputs to the operators guarantees that the algebra is composable. The `Summarize` operator provides the foundation for defining the coherence measure between source data and its new target application. The `Match` operator serves to define the similarity measure between information sources. The remaining operators of the algebra generalize characteristic applications of semistructured data that are currently under investigation. Unary operators all take a coherence measure, while binary operators require a similarity measure. Note that these measures do not define a *metric*, but they do estimate the level of relevancy and similarity between data sources and target applications. Operators we have used for one particular source are presented in detail in

Chapter 4.

The novelty of the algebra is threefold. First, we have decoupled the selection of coherent parts of the source data from the determination of the articulation points between complementary information sources. For example, the relevance of the terms 'arrival' and 'arr.' in the airline meal survey is computed using a coherence measure. A separate computation determines that those terms are similar and form an articulation point between the two sources for the airline meal survey. Second, the coherence and similarity measures are created, refined and maintained in an iterative process using the algebra rather than a separate language. We bootstrap the development of the nexus, for instance, by using the operators of the algebra itself, rather than manually creating it. In contrast a system such as CYC contains an inference system which has never been used to build, maintain or support the ontology that CYC uses in its inference. Third, we selected operators of the algebra to mirror classes of applications of semistructured data, rather than low level abstract primitives which are difficult to compose into meaningful operations. However, the algebra still retains the ability to compose and reorder its operations according to formal rules. For example, one of the most common operations performed on an information source is the computation of all of the unique terms in the source. The `Glossarize` operation of the algebra was specifically designed to support that type of computation. In this way, the operations are intuitive and have immediate value to application experts building articulations.

2.1.1 Primary Source Data Sets

Typical Web sources are semistructured, that is, they contain well defined structural elements, but do not have a fully regular schema. There is a growing literature on generating wrappers for individual semistructured data sources [MMK98], and on the subsequent access of the data through a query interface [LRO96]. Also, there is some research in the area of view maintenance of these wrapped sources. However, the wrappers seen today are handcrafted using languages separate from the query language, and the primary assumption for view maintenance is that the source structure and formatting is stable. In practice, handcrafting wrappers is only feasible as long as the total number of data sources is small, and as long as the sources themselves are fairly stable. On the other hand, when sources change frequently as on the World Wide Web, it becomes difficult to fully maintain more than a few sources. Experience at mySimon.com [myS99], a successful comparison shopping website, showed that at any given time one quarter to one third of the manually constructed wrappers were not functioning correctly. To further complicate matters, when the data source is large as well as irregular, handcrafting the wrapper in itself becomes an onerous task.

Our initial experiments were with an on-line version of the 1913 Webster's dictionary that is available through the Gutenberg Project [PRO99b]. The original dictionary is a corpus of over 50 MB containing some 113,000 terms, 15,000 of which are variant spellings and synonyms. Altogether, the definitions contain over 2,000,000 words. The source data of the dictionary was originally scanned and converted to text via character recognition software; it therefore contains thousands of errors and inconsistencies. Having to deal with errors helps our scheme to achieve the robustness needed for real-world settings. Our target application for the dictionary data is the construction of a graph of the definitions from which we can determine related terms and automatically generate thesaurus entries. Misspellings and incompleteness in the terms and definitions, as well as errors in the labeling of the data, resulted in over five percent of the data being incorrectly interpreted using a naive wrapper. In this case the source inaccuracy affected over 100,000 arcs in the graph, and about 4,000 of the graph's nodes.

Subsequently we developed an entirely new repository based on the Oxford English Dictionary, Second Edition (OED) [Tom99], a corpus of 570 MB. The resulting structure, at over 510,000 nodes and 8,000,000 arcs, is four times the size of the Webster's repository, and demonstrates the scalability of our approach. One strength of the system is the ability to iteratively refine a simple wrapper using an operator from our algebra to reduce the exception rate below one percent. We will present the OED and wrapper refinement in Chapters 3 and 4.

Bringing similar information together from multiple data sources introduces the problem of semantic heterogeneity. The data in an individual source serves a purpose originally defined by the source's maintainer. This purpose is partially spelled out in the schema and structure of the data set. The applications which use the data usually express implicit assumptions about the data, which are not contained in the data itself. When gathering information from multiple sources to apply to a new use, it is necessary to explicitly resolve differences in the semantics of the data. It is formally sufficient to resolve only the differences that are germane to the new context to which the data is to be applied. Unfortunately, there are no exact computational methods for recovering implicit semantics from multiple data sets. Instead, we developed semi-automatic techniques for exposing incorrect, incomplete and inconsistent data within a source. Such semi-automatic techniques reduce the burden of manually deriving the source semantics that are relevant to the new applications of the source data.

We find an example of substantial source heterogeneity from the NATO [NAT99] Web pages starting at: `http://www.nato.int/`. This site contains information about the NATO alliance, as well as the Partnership for Peace alliance, and contains links to various pages from their member nations. In particular, two pages provide links to web pages maintained by the governments of the members of these two organizations. On the surface, these pages would appear to represent similar information, but they revealed substantial heterogeneity when we used them in combination. Beyond the differences in language of expression, it was necessary to understand the types of differences in the data in order to work with the data we collected. Using the algebra's `Match` operator allowed us to iteratively define conditions for combining NATO member government data from their Web sites. The results show that our initial specification of the articulation obtains 70% of the matches that a human expert would define, without adding any incorrect matches.

Since we assume sources are autonomous, they may change at any time. We are able to separate changes of a source that affect their relevancy to our application from those changes that affect their similarity to other sources with which we combine them. These definitions form the underpinning for the formal definitions of the coherence and similarity measures. We begin by pinning down a definition of ontology, domain context and articulation.

2.1.2 Ontologies

The term ontology is used to mean different things by almost every researcher who works with ontologies. We have developed a hierarchy of definitions that cover the most common usages of the term [DJ00]. In this dissertation, however, we will stick to a very simple mathematical definition of ontology, given below:

$$M = (S, r_1, ..., r_n, m^*) \tag{2.3}$$

An ontology M is a tuple, containing a set S, whose elements are objects. M also contains one or more relations r_i ranging over the objects in S, and a mapping m^*, whose domain are objects in S and whose range are items and concepts in the external world. We provide a small example to show a minimalistic ontology of pets. Here the set of objects

S is given in brackets. The relation r indicates that there are three kinds of pets (**dog**, **cat**, **goldfish**), and that two of those are mammals (**dog**, **cat**), while the other is a fish. The mapping m^* simply indicates that we can associate each object in S with the intended real world equivalent having the same name. We assume that the capitalized terms here refer to an actual real-world concept.

$$M_{pet} = \begin{array}{rl} (S: & [dog, cat, mammal, goldfish, fish, pet], \\ r: & dog \sim pet, cat \sim pet, goldfish \sim pet, \\ & dog \sim mammal, cat \sim mammal, goldfish \sim fish \\ m: & dog \mapsto DOG, cat \mapsto CAT, mammal \mapsto MAMMAL, \\ & goldfish \mapsto GOLDFISH, fish \mapsto FISH, pet \mapsto PET) \end{array} \tag{2.4}$$

The mapping m^* may be implicitly defined in the object labels or *terms* applied to the elements of the set, or may be computable from the relations r_i between object items. In the above example the object labels do indeed map directly to a real world equivalent. If, however, the object label for **goldfish** were **gf** we could perhaps still deduce **gf** $\mapsto$ **GOLDFISH** from the facts in r, namely, **gf** $\sim$ **pet** and **gf** $\sim$ **fish**. The mapping m^* is not directly representable, since its range contains external world objects, but it is critical to the definition. We may write a *proxy function* or ruleset m, as we've done above, to stand in for the mapping m^* when we agree on the terms that represent the external objects. From this standpoint, we see that an ontology represents an agreement to define the relationships between external world objects in a certain form, and the mapping m^* is the *vocabulary* that represents that agreement. The proxy function m mentioned above is equivalent to an interpretation function as defined in the field of logic [GN88]. Given the above definition, we now clarify the notion of ontological domain. Note that we distinguish ontological domain from mathematical domain as used above. In the text that follows we will use domain to mean ontological domain. A domain is simply a vocabulary that represents a set of external objects consistently. We typically use the domain vocabulary as our proxy function m in an ontology, because each term in the vocabulary is used to represent one object from the external world. We will also consider n-tuples of terms from a domain to be part of the domain. Referring back to the example above, tuples such as (**dog**, **cat**) belong to the domain of M_{pet}.

The above definition of ontology illuminates the need for articulations. When we want to use an existing ontology for a new application, or combine terms from more than one

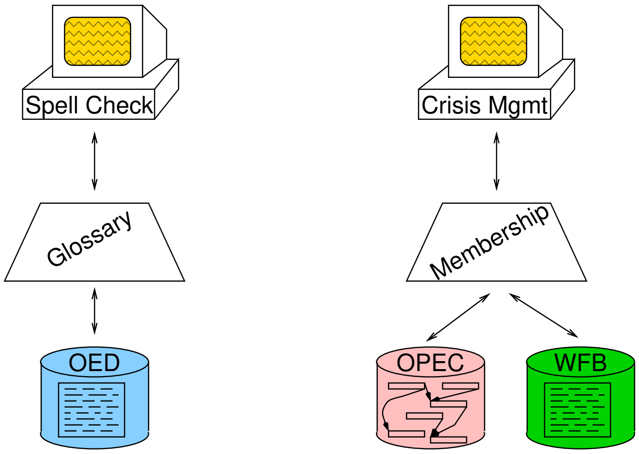


Figure 2.1: Unary and Binary Articulations

source ontology for use in an application, the semantic mappings, or more properly the proxy functions associated with the ontologies, must undergo a transformation. If we want to use M_{pet} to cover the type of food the animals eat, we see that we can transform **mammal** to **carnivore**, and **fish** to **vegetarian**. Note that this transformation is only valid for the specific data of M_{pet}, and the addition of **piranha** would require a different transformation. Articulations embody this transformation and enforce access to a controlled portion of one or two source ontologies, using the vocabulary of a target application. An articulation is one or two functions from source domains to a target domain, associated with composition rules over the ontologies' object sets. We use the functions to describe how a vocabulary is transformed into a distinct target vocabulary. Each term transformed by the function is associated with composition rules that describe how the items in the ontology are viewed through the articulation.

Formally, our articulation A_t is a partial transform of possibly multiple source ontologies $M_1, ..., M_n$. This transform provides a new proxy function m_t that maps only a portion of the objects, relations in the source ontologies. The new mapping preserves only the objects that are relevant to a target application t.

$$M_t = A_t(M_1, ..., M_n) \tag{2.5}$$

The equation above expresses the most general case where there are n source ontologies, but we will restrict ourselves to the unary and binary cases only. Note that we consider both unary and binary articulations where unary and binary refer to the number of source ontologies. We do not consider n-ary articulations separately as they can be defined by the composition of multiple binary articulations. Articulation is needed not only to adapt sources to operate together, but also to adapt sources to the requirements of a new application for which the sources were not originally designed. We introduce a new ontology, $M_{kennelclub}$, in order to show a sample articulation with M_{pet}.

$$M_{kennelclub} = \begin{array}{ll} (S: & [kennelclub, dog, bitch, hound, toy, terrier], \\ r: & hound \sim dog, toy \sim dog, terrier \sim dog, \\ & hound \sim bitch, toy \sim bitch, terrier \sim bitch, \\ & dog \sim kennelclub, bitch \sim kennelclub \\ m: & hound \mapsto HOUND, toy \mapsto TOY, terrier \mapsto TERRIER, \\ & dog \mapsto DOG, bitch \mapsto BITCH, kennelclub \mapsto KENNELCLUB) \end{array} \tag{2.6}$$

Figure 2.2 is an example of an articulation which blends the objects of M_{pet} and $M_{kennelclub}$ above. Note that **dog** in M_{pet} matches two objects in $M_{kennelclub}$, and that these are the only true matches between the two source ontologies. This simple matching produces a new ontology which blends the two sources without filtering out any of the objects. We will see in Section 2.1.4 the `connect` rule which we use to make the matches between the two above ontologies explicit. It is important to note that it is common for articulations to reduce the size of the source ontologies that they process.

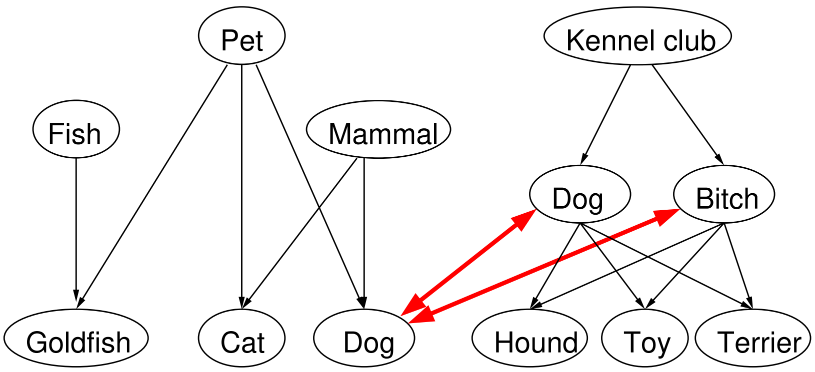


Figure 2.2: Dog Articulation

As examples, Figure 2.1 illustrates the two types of articulations we consider. The unary type of articulation benefits a spell checking application by glossarizing the defined terms in the Oxford English Dictionary. In contrast, a crisis management application can use a binary articulation that combines current OPEC membership information from the OPEC web pages with geographical and political information about OPEC members taken from the World Factbook (WFB) [Cen97]. The following sections cover the definitions of the objects which are members of an ontology and the composition rules that allow us to construct articulations.

2.1.3 Object Model

Rather than invent yet another model to represent the objects of our ontologies, we simply allow any model that satisfies the semantic abstractions listed below. We call our instance of this model *Atoms Modeled in Objects* or AMO for short. An object O is a tuple containing as its arguments an identifier r, an attribute set s, and a sequence of labeled atoms $a[\,]$.

$$O = (r, s, a[\,]) \tag{2.7}$$

These object semantics are general enough to subsume some existing models, and are powerful enough to simulate others. In particular, it is easy to import XML documents and convert OEM objects to this model. As specified, objects do not explicitly type their arguments, because none of the material in this dissertation depends on object typing. In short, any object model that is adequate for our purposes provides for the following abstractions:

Reference

object identity

Atom

object references, strings, numbers

Attribute set

labeled set of atoms

Value

sequence of atoms and sub-objects

In addition to its attribute set, an object has a value which is an ordered list of atoms and nested objects. Since an object value is a sequence, *position* of its subcomponents is an important part of the value. Any atom or reference in an object value is indexed by its position in the object. A reference is an object identifier (OID), and all objects contain an OID attribute, that is unique across all objects. Also, the label attribute of an object is optionally set. References from an object to others represent local identifiers of the referred object. Indeed, they can only be used to access the referred object from the objects which contain them, and can form part of a *path expression* to reach the referred object. References without a source object are OIDs which are considered global with respect to the domain of the information source. Reachability of objects through repeated traversal of references in the AMO model is the sole criterion of object existence. The set of objects reachable from a given object constitute its universe. The set of objects reachable from OID (globally) referenced objects constitute the universe of a domain. Figure 2.3 shows each of the above abstractions' graphical representation. The arc pointing to the object is the OID, the small white rectangles are the atoms which together represent the value of the object. The first atom represents the object's attribute set, the second and fourth are strings, while the third atom is a number. The fifth atom is a reference to a sub-object of the object. Changing the order of any of the atoms changes the value of the object, as much as changing the value of any of the atoms. Below we present primitive operations over AMO which relate the different parts of the model.

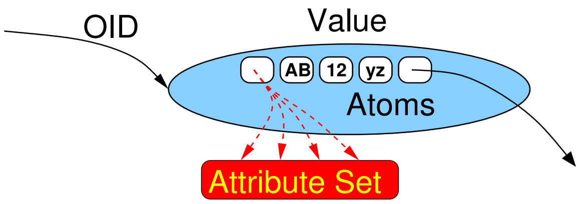


Figure 2.3: AMO Object Model

Note that only objects have an identity to which others can refer. All other components of the model are atomic values, which may not be shared, and have no existence independent of the object that contains them. Atoms may, however, be copied, edited and reified. Objects have a value, which is defined as the sequence of values that compose it. This model corresponds to XML supplemented with object identity, where a bracketed `<obj>...</obj>` XML object corresponds to an object in our model, and its URI is its identifier. This style of object access corresponds very closely to applying a document object model (DOM) to XML. The object model allows us to represent, without modification, HTML and XML documents, data in Stanford's OEM [GCCM96] format, plain text, and database relations. It is simple to convert plain text to AMO, by simply wrapping the text in a single object where the entire text is the sole atom. Furthermore, AMO is rich enough to represent more complex relationships such as inheritance and typing. This expressiveness allows it to simulate UML [Fow98] and frame [Kar92] models.

2.1.4 Rule Language

While the algebra we've alluded to in Section 2.1 transforms contexts and the object graphs they represent, it is also convenient to be able to express these transformations specifically in terms of operations on the objects that constitute the graphs. These primitive operations support the algebra and form the rule language AMORL, out of which we construct the similarity and coherence expressions defined below. These expressions are constraints to the algebraic operators, as selection and join conditions are in the relational algebra. In essence, AMORL performs the low-level syntactic transformations on components, in support of the semantic operations of the algebra.

The operators listed below form expressions that operate on an idealized data stream from the information source to the application. Expressions concatenated together are executed in sequential order, and require pipelined execution or multiple passes of the data stream, while nested expressions represent a single pass of the data stream.

In addition we've listed a few sample rules taken from the articulation that defines the Webster's dictionary nexus. Note that these rules apply equally well to AMO objects as they do to plain text, since plain text can be considered an atom of a single enclosing object. Also, these rules specify portions of objects or atoms based on regular expression *patterns*. This object addressing style allows us to refer to zero or more objects, depending on the specificity of the expression.

A pattern identifies one or more objects in an object graph by value or by reference. When a rule matches objects, the transformations are determined by the other parameters of the rule. OID sets, atom sets and position sets that serve as rule parameters contain the values that modify the objects that match the pattern. The types of pattern predicates are:

- path expressions over references
- set membership expressions over attributes
- regular expressions over strings

- relational expressions over numbers

Note that the last three pattern expression types refer to values contained within an object, whereas the first expression type refers to any path that reaches an object. Given a classification of vehicles by the medium in which they operate, and their purpose of use, `vehicle.land.recreational.*` is a path expression that will reach objects such as bicycle, skates, skis, and `vehicle.*.cargo` reaches freight train, tanker ship, and cargo plane. Membership expressions appear in path expressions, as well as on their own and are combined disjunctively or conjunctively. Assuming that vehicles have attributes `wheel`, `license` and `motor`, then the *disjunctive* pattern `vehicle.land.recreation.*.[wheel|motor]` can return in-line skate or snowmobile, which have either attribute. In contrast, the *conjunctive* expression `vehicle.air.recreational.*.[wheel&license]` returns piper cub, but not hang glider (since hang gliders don't have wheels).

Note that the conversion rules defined below allow transformations from values to objects (reify), which enables their use in wrapping sources which are not already within the object model. These conversion operators also allow the uninterpreted substructure of an object to be transformed into attributes of the object. Such refinement of structure enables us to recognize objects with differing granularities as being similar.

- constructor

`create`

generate new objects to *proxy*, or stand in for and refer to existing objects. Takes an object pattern and a set of references as parameters, to create objects based on the pattern.

create('England', {'English'})

add the proper noun *England* to the articulation and have it reference the term *English*.

- connectors

`connect`

generate new objects to proxy for and equate distinct objects from separate sources. Takes two object patterns and an OID set as parameters.

`inherit`

generate new proxy objects in a subsumption relationship between separate sources. Takes an existing OID, an object pattern and an OID set as parameters.

- editors

`insert`

insert an atom into object at a specified index position. Takes an object pattern, an atom set and a position set as parameters.

`edit`

edit an atom within object at a specified index position. Takes a source pattern, a replacement value, an atom set and a position set as parameters.

edit('&ccdil\.', 'c', *)

replace French ç with *c* because it is not uniformly applied throughout the document

`move`

move an atom within object to a specified index position. Takes a source pattern, a destination value, an atom set and a position set as parameters.

`delete`

delete an atom from object at a specified index position. Takes an object pattern, an atom set and a position set as parameters.

- converters

`reify`

replace an atom in an object with a reference to an object containing that value. Takes an object pattern and a position set as parameters.

reify(edit('ie[drs]$', 'y', *), *)

plurals of words ending in 'y' are returned to singular form to match with dictionary entries

`fuse`

replace the reference(s) in the referring object with the values of the referred object in the appropriate index position. Takes an object pattern and a position set as parameters.

Strictly speaking, the set of operators above is not minimal. It is simple to simulate the `inherit` object, `edit` value, and `move` value rules using the other rules.

Note that a context is itself an object and that the rule language specifies how the context is populated with values from sources. Constructors create new objects which are not represented directly in sources. Connectors generate proxy objects that stand in for one or more objects from sources, which may then be modified using editors and converters. In the following section we list the algebraic operators without going deeper into their definitions.

2.1.5 Algebraic Operators

The algebra consists of a composable set of operators that transform contexts into contexts. These contexts, defined in more detail in Section 5.0.1, encapsulate ontologies with a guarantee of semantic consistency. The operators are listed below with a brief description of the operation they perform. The abbreviated form of operator names given here is also used in the dissertation.

- Unary operators

  S `Summarize`

  term classification

  G `Glossarize`

  listing of terms

  F `Filter`

  object instance reduction

  E `Extract`

  schema simplification

- Binary operators

  M `Match`

  term corroboration and refactoring

  D `Difference`

  schema distance measure

I `Intersect`

schema discovery

B `Blend`

schema extension

Unary operators reformulate source information with respect to the requirements of the target application. `Summarize` is the canonical unary operator. It is used to establish and refine a context within which the source knowledge meets the requirements of the application. For example, S groups a set of objects by an attribute's value, by the attribute's presence or absence, or by objects' current path expression. Binary operators express the linkages between sources that are germane to the needs of an application. Playing the same role for binary operators that S plays for unary operators, `Match` or M is the canonical binary operator, and is used to develop and refine articulations between sources. M groups objects in two sources together when they have similar attribute values and path structure. We define the operators in greater detail in Chapter 5, for those wanting a more complete overview of the infrastructure supporting the claims of the thesis. These claims are the subject of Section 2.2, which led us to the hypotheses we defined in Section 1.2.

2.2 Thesis Contributions

As we saw in Section 1.2 we have three hypotheses to confirm in the dissertation. This chapter has provided the background to describe our contributions in more detail and to introduce the key chapters of this dissertation. Although database systems typically allow a very rich query capability over the data they maintain, there is a deep assumption that the data conforms strictly to its schema. This assumption is so pervasive in the database field that it is very difficult to present work under different conditions. In this dissertation the paramount assumption is that it is difficult to bring information from multiple sources into a single consistent format. Even stronger, it is non-trivial to refactor a single large information source for a new purpose, regardless of its level of consistency for its original application. The scope of this thesis does not permit us to build a complete query answering system; we focus instead on this lower level problem of restructuring information.

2.2.1 The Word Nexus

The nexus is a repository of relationships between terms contained in an on-line dictionary. With this new graph structure it becomes possible to consider a large number of relationships over the dictionary terms, and to contemplate the feasibility of computing them from the structure itself, rather than manually extracting them. First off, is it possible to determine the most important definitional terms in the dictionary? More conservatively, can we find for a given term, which are the words in its definition that contribute most to its meaning? Also, how are similar terms related in the graph structure? As it turns out, new algorithms adapted from graph theory point in the right direction. With these algorithms the dictionary repository itself plays a central role in facilitating the development of other articulations over other information sources.

2.2.2 Nexus Extensions

We extend the word nexus by computing new structural relationships between directed labeled graphs. Structural relationships refer to sets of arcs in a graph that may share an end point, or have end points with related labels. Considering the entire graph as a flow network, we look for steady states of flow across the arcs between the nodes. This flow represents the importance, with respect to usage, of each term in the dictionary. Once the notion of network flow over the graph exists, we define further relationships. For example, similar terms in the dictionary should have similar definitions. We define a new algorithm for computing similarity of terms in a large repository. Using the flow between nodes it becomes possible to quantify what similar means. With these relationships the cost of computing similar terms across other repositories is reduced to a linear time operation.

2.2.3 The SKEIN System

SKEIN is our framework for rapid development and maintenance of rule sets that mediate heterogeneous information sources. We show how the object model makes it easy to establish new structures given a model of the source information. We define how the rule language supports the restructuring of the data as inconsistencies and other irregularities become evident in the data. Taking an on-line Webster's dictionary as our example we create a 97,000 node directed labeled graph from the dictionary definitions. We see the adaptability of the rules creating this graph when the underlying data source is updated. We reuse the rules to transform two other related data sources. First, a Roget's thesaurus, with one thousand nodes in the resulting graph, demonstrates resilience across data sources. Second, the Oxford English Dictionary, with over 327,000 nodes, shows the scalability of the rule system.

2.3 Groundwork Review

In this chapter we defined the important terms of the dissertation. We presented a simple object model and rule language which form the foundation of an algebra, and are also prerequisites for constructing structures such as the nexus. We also very briefly introduced the notion of algebraic operators in our system. In the following chapters we treat each of our thesis contributions in turn, and show how it fulfills one of the hypotheses we defined in Section 1.2.

Chapter 3

Nexus Development and Maintenance

Chapter Outline

This chapter investigates the refactoring of existing knowledge bases for uses that were unanticipated by their original creators. The extended example of Chapter 1 shows how general-purpose airline flight schedules can be refactored to provide information about meals on airline flights. In our model knowledge sources are assumed to be autonomous and are not changed by the refactoring process, but may change as they are updated for their original application domain. The techniques developed for refactoring knowledge bases are therefore also applied to the maintenance of such refactored information when the underlying knowledge sources change. The autonomy of diverse knowledge sources is an obstacle to integrating all pertinent knowledge within a single integrated knowledge base. The cost of maintaining integrated knowledge within a single knowledge base grows both with the volatility and the number of the sources from which the information originates. Establishing and maintaining application specific portions of knowledge sources are therefore major challenges to ontology management.

Rather than materializing all of the information from the sources into a single knowledge base, we present algebraic operations in Sections 3.2.1 and 3.2.3 that enable the construction of virtual knowledge bases geared towards a specific application. Operators express the relevant parts of a source and the conditions for combining sources using AMORL defined in Chapter 2. Rules which expose the relevant parts of a source determine what we defined in Section 2.1 a coherence measure between the source and its target application. The rules which articulate knowledge from diverse sources establish a similarity measure between them. The sections that follow describe our selection of our first large data source to analyze and how we reused and extended our results to other data sources.

3.0.1 Basic Nexus Characteristics

This chapter presents a case study of repository construction from an autonomous source, for use in a real world application. In the course of developing the application, the underlying data has been updated three times, requiring repeated execution of a source refactoring operation to bring the repository up to date. We show how a principled approach based on algebraic operators has made it possible to create and maintain access to this information with a low overhead cost, thus fulfilling the first hypothesis of this dissertation.

Proprietary dictionaries such as the Mirriam-Webster [Mir99] and encyclopedias like Encyclopedia Britannica [Enc99] are available on the World Wide Web. Typically they present a user interface that provides access to one term at a time in the typical case. Most freely available dictionaries have partial coverage, or are limited to a specific domain [Ger96]. Among the most extensive freely available corpora we find WordNet [MBF^{+}90], which has been hand-crafted over several years by interested specialists, and does not claim to be a complete language reference. However, some sources with complete and relatively unbiased coverage do exist. The on-line version of the 1913 Webster's dictionary [MIC96] is available through the Gutenberg Project [PRO99b].

The source data of the dictionary was originally scanned and converted to text via character recognition software and therefore contains thousands of errors and inconsistencies. The abundance of errors in the dictionary data makes it an ideal test bed for our research. Dealing with incorrectness in sources provides our approach with the robustness needed for real-world settings. The second reason the Webster's dictionary is a valuable source for study is that it is autonomously maintained and updated. At roughly semi-annual intervals its maintainer adds material to it and corrects some of the errors in the text. The purpose of the updates is to eventually bring up the content up to date. Our target application for the dictionary data is the construction of a graph of the definitions, forming a dictionary repository, from which we can determine related terms, and automatically generate thesaurus-style entries [Jan99b]. Accuracy in the data is important for meaningful results, since we run flow algorithms on the graph structure. Misspellings and incompleteness in the terms and definitions, as well as errors in the labeling of the data resulted in over five percent of the data being incorrectly interpreted using a naive wrapper. We iteratively refined the naive wrapper using operators from our algebra to reduce the exception rate below one percent. For comparison, our final articulation requires just under 500 rules performing some 300,000 transformations in total. These transformations handle approximately 80,000 inconsistencies representing approximately 4% of the Webster's source data. Of these rules the majority are executed exactly once. The great benefit is that the number of rules is so many orders of magnitude less than the size of the resulting nexus.

The Webster's dictionary is autonomously updated as part of an ongoing effort to remove its inconsistencies, and to add updated material. This updating makes it possible to verify our maintenance claims for the repository we have constructed. Our use for the information contained in the dictionary is typical of a demanding application, as it provides a legitimate service and it has strict tolerances for how much erroneous information it allows. We discuss, within an algebraic framework, how we establish and maintain a context that takes the source data and makes it available to the application, meeting the application's requirements. We present the initial derivation of the context for our application, and show in the following section how the refinement process is repeated, albeit with less overhead, when the source undergoes change. In the final section we describe our application, and discuss our future directions of research.

In the next section, we describe the dictionary repository, as well as its creation and refinement. We focus on four operators used in this process, the `Glossarize`, `Extract`, `Match` and `Summarize` (G, E, M, S, respectively) operators. The G operator generates the objects in the repository, while E associates a sequence of terms with each object, and M generates references between objects based on the sequences created by E. The S operator is the primary tool for refining and maintaining a context between a knowledge source and an application.

3.1 Webster's Dictionary

This section describes the source data, the on-line Webster's dictionary, that we use in our experiments. This edition of Webster's dictionary was recently (1996) converted to text format from scanned images using OCR (Optical Character Recognition) The resulting text is tagged using SGML style tags to mark the parts of the definitions. Definitions from the

```
<p><hw>E"go*ism</hw> <pr>(?)</pr>,
<pos>n.</pos> <ety>[F.
<ets>\'82go\'8bsme</ets>,
fr. L. <ets>-ego</ets>
I.  See <er>I</er>, and
cf. <er>Egotism</er>.]</ety>
<sn>1.</sn> <fld>(Philos.)</fld>
<def>The doctrine of certain extreme
adherents or disciples of Descartes and
Johann Gottlieb Fichte, which finds all
the elements of knowledge in the
<xex>ego</xex> and the relations which
it implies or provides for.</def></p>

<p><sn>2.</sn>
<def>Excessive love and thought of self;
the habit of regarding one's self as the
center of every interest; selfishness; --
opposed to <xex>altruism</xex>.</def></p>
```


Figure 3.1: Webster Definition of Egoism

dictionary data for Egoism are shown in Figure 3.1.

We have been using the dictionary data to research the relationships between dictionary terms. One potential application of these relationships is the extraction of thesaurus style entries, such as for the term Egoism in Figure 3.2. The terms in the figure all have dictionary definitions that contain other terms in the figure. Whenever there is an arc between two nodes it indicates that the definition of the first term contains the second term. For example, the term Altruism contains the term Egoism in its definition, and as it turns out the reverse is also true. The relationships between terms are expressed by extracting the implicit structure contained in the dictionary, rather than explicitly marked ones such as synonyms. This implicit structure is found by positing that terms must be connected to the words used to define them. Note the strong relationships between terms in the graph. Most of the terms in the cluster depicted in Figure 3.2 are synonyms or near synonyms of Egoism, and the others are antonyms. We've defined as coherence the relationship that connects these terms, that is, the degree of relevance to each other that the words share. In the SKEIN system we use the coherence relation to discover which terms in an information source are

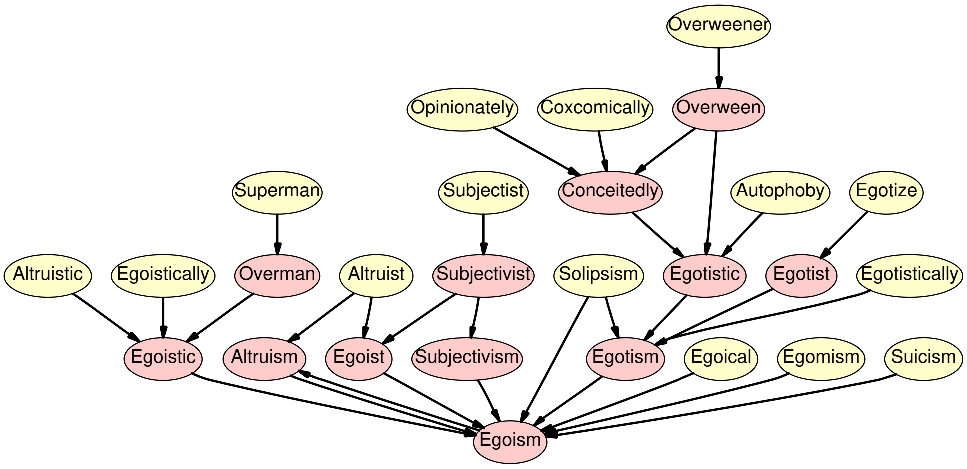


Figure 3.2: Automatic Thesaurus Extraction from Dictionary

relevant to the application for which we are building a mediator. We call nexus the tool we build with all of the collected dictionary data. Its purpose is to serve as a tool in reducing the occurrence of lexical mismatch when enabling diverse ontologies to interoperate. We describe this use of the nexus in Chapter 5.

The dictionary is organized as a set of head words, associated with a pronunciation, etymology, possibly multiple definition entries, quotes and other tagged items. For the purpose of our application we were interested in the head words `<hw>...</hw>` and definitions `<def>...</def>`. The special meaning of many other types of tagging, such as cross references, `<xex>...</xex>`, are ignored. While it is certain that these ignored tags would add relevant terms to the structure we wish to extract. However, preserving the terms in these other tags would make it difficult to verify the hypothesis that the definitions contain words that are strongly related to the terms they define. Since part of our research aims to show the value of the nexus we extract from the dictionary, it is best to measure the value of the nexus without additional terms that would complicate the measurement. Head words and definitions are in a many to many relationship, as each definition refers to different senses of a term, and each head word has variant spellings. We constructed a directed graph from the definitions as follows:

1. each head word and definition grouping is a node

2. each word in a definition node is an arc to the node having that head word

Substantial manipulation is required to bring the dictionary data into a format ready for generating a graph [JW99]. Other problems in the transformation process are listed below.

- syllable and accent markers in head words (E"go*ism)
- mis-tagged fields
- misspelled head words
- stemming and irregular verbs (Hopelessness)
- common abbreviations in definitions (etc.)
- undefined words with common prefixes (un-)
- multi-word head words (Water Buffalo)
- undefined hyphenated and compound words

Markers such as those indicated here in `E"go*ism` are removed first. We discover fields that are missing an end tag when head words are inordinately long, or definitions are missing, or contain other definitions inside them. Misspelled head words are recognized when the definitions consistently use a variant form of the term. The easiest type of misspelled head word to identify are compound words, others contain sequences of characters that do not occur in English. Definitions use many forms of a word that do not appear as a head word. In particular, word forms that contain multiple suffixes rarely are head words, so we need procedures to convert definition words to head words. Likewise, abbreviations, proper nouns, and words with very common prefixes are infrequently defined. Finally, hyphenated words and compound words are very inconsistently defined and used. Thousands of cases occur where words are defined with a hyphen and used without, and vice-versa.

When a conjugated verb form appears as a head word we use it for generating graph arcs. Otherwise we stem definition words until we find a head word that matches. Also, whenever we find instances of a multi-word head word in the definitions, we prefer it over the individual words for generating a graph arc. Since words often appear multiple times in a single definition we allow multiple arcs between graph nodes. Dealing with undefined terms and spelling errors is the most complex issue in the graph generation, and accounts for the quasi-totality of the structural errors in the graph.

Table 3.1: Rule Categories and Frequency of Occurrence

- Domain Universals: $1^{-1} - 1$

  edit('&ccdil\.', 'c', *) : replace French ç with *c*

- Language Related: $1^{-2} - 1^{-1}$

  reify(edit('ie[drs]$', 'y', *), *) : plurals of words ending in 'y'
  reify(edit('[Dd]r\.', 'doctor', *), *) : abbr. of doctor

- Domain/Language Interaction: $1^{-4} - 1^{-3}$

  create('England', 'English') : add proper noun to dictionary

- Domain Idiosyncratic Instances: $1^{-6} - 1^{-5}$

  edit('aconitiq', 'aconitia', *) : spelling error in dictionary head word

To give a better sense of the types of rules we used and how frequently they fire, we've listed a few of the rules in Table 3.1. Frequencies are listed relative to the number of terms processed in the sources. Domain universals are the rules which apply to terms that appear frequently throughout the data source. Note that these terms are universal to the specific source domain, and rules developed to change them are not generally valid in other contexts. Language related rules are those that morphologically transform terms into a core form. For example, transforming a plural into a singular form or a past tense into a present tense, are examples of language related rules. There are also rules to handle cases where the characteristics of the domain interact with the language. In our example, we developed rules to handle situations where proper nouns were used in definitions, even though they themselves are not defined in the dictionary. Finally, a category of rules handle domain idiosyncratic problems, such as spelling errors. We have found that domain idiosyncratic rules make up the majority of those in our articulations, but that the domain universals are those that fire most often. We've listed the categories in Table 3.1 in their order of frequency of occurrence.

Even after accounting for accented characters, a naive script is unable to properly assign over five percent of the words, because of the above mentioned differences between the actual data in the dictionary and its assumed structure. Any errors in the computation of the graph would affect any subsequent computation of related terms for the thesaurus application. Therefore, we set a goal of 99% accuracy in the conversion of the dictionary data to a graph structure.

3.2 Repository Construction

In this section we show how we use the algebraic framework to rapidly generate a large object repository from a text source. The product of the algebraic operators are articulations, the rulesets that generate the repository. We show that without complete knowledge of both sources and target application, iterative construction of articulations is necessary.

3.2.1 Extraction from Source Data

In this section we cover the application of G, E and M operators to the dictionary source. The actual code for these operations can be found in Appendix B.

The `Glossarize` operator G takes source data in AMO format and returns a list containing all of the objects in the source data. Recall from Section 2.1.3 that plain text converts trivially to AMO. In fact G can contain a `reify` rule that explicitly performs this conversion. The effect of G is to flatten any complex object structure in a source, because all sub-object relationships are removed from the resulting list of objects. In the case of the Webster's dictionary, there are no objects in the source initially. We use AMORL to reify objects from the raw data. The sequence is implemented in Table 3.2. The code we use to perform the extraction is presented in Section B.2. The method of generating proxy objects to represent the raw source is as follows:

1. segment data into *chunks* defined as containing a minimum of:
   - one head word (marked by `<hw>`, `</hw>` tags)
   - one definition (marked by `<def>`, `</def>` tags)
2. merge chunks if head words match a pre-existing chunk
3. reify chunks as objects then enumerate them
4. label objects using head word value

The `Extract` operator E takes source data and generates objects whose contents are a list of primitive values. In the case of the dictionary, only the portion of the source contained within `<def>`, `</def>` tags is used. From the definitions we generate one list of string values for each glossarized object of the nexus. This step of extracting the list of string values is the central part of the `E` operator. We list the code we use to perform the extraction in Section B.3. The method of extracting the definition portion of the proxy objects defined above is as follows:

1. from each object chunk, select the `<def>`, `</def>` tagged data
2. generate a list using the white-space separated terms in the definitions
3. replace the current object chunk's value with the newly created list

The line marked `(2)` of Table 3.2 shows the AMORL operations to compute `E` over the unstructured object chunks created by `G`. Finally, `Match` takes the value of each object (the list computed above by `E`), and determines for each element of the list which object best matches it. The `M` operator substitutes into each glossarized object a list of references to other objects for each list of strings extracted by `E`. We perform the match using code listed in Appendix B.5.

3.2.2 Constructing the Coherence Expression

The dictionary nexus is constructed from the raw corpus data of the dictionary as obtained from the ftp site. We proceed by writing the transformation as a coherence expression. The simplest form of the script that represents such an expression consists of the following AMORL operations. Table 3.2 shows the first cut of the expression. The key lines in the table are `(1)` that generates the objects for `G` and `(2)` that extracts the sequences of definition terms for `E` given the Webster's dictionary source data. Regular expressions are simplified for conciseness of presentation.

This script creates an object that represents the entire source, which is then subdivided into chunks containing at least one head word and one definition, which are then extracted into separate sets within the chunk. Each script fragment such as Table 3.2 represents operations in the course of a single pass through the source data, the output of which may be passed to another script. This initial script very approximately expresses the coherence relationship between the dictionary and the thesaurus application. The final script, containing about 500 conversion operations that perform over 300,000 transformations, is presented in Appendices B.2 and B.3. The script remains associated with the output, and is

Table 3.2: Coherence Expression for G and E Operations

```
                        // assign contents to object
    dictionary = create(".*") {
                        // generate dictionary entries
      entry = reify("<hw>\(.*\)<hw>") {
                        // insert head word values
(1)     head_word = insert("<hw>\([^<]+\)</hw>") {
                        // remove syllable and accent markers
          self = edit("[\"‘*]", "")
        }
                        // insert definition object
(2)       definition = reify("<def>\(.*\)</def>") {
                        // insert definition word attributes
          word[] = insert("\([^ ]+\)[ \n]")
        }
      }
    }
```

part of the definition of the context which represents that output. In the following section, we show how the S operator allows us to capture some of the classes of conversions which enhanced the above initial script.

This articulation represents the first large scale example that we developed for our thesis research. It demonstrates parts of the algebraic framework that we use to structure the dictionary nexus and to make our system scalable. Here we present a definition for the `Summarize` operator and show its use in refining the repository.

3.2.3 Summarization of Exceptions

This section shows how we take the results of one iteration of the algebraic operation, determine those results which do not match application requirements, and aggregate them to find classes of exceptions.

The `Summarize` operator is a unary operator that transforms source data according to a predicate which corresponds to a coherence expression. The predicate therefore consists of a ruleset from the rule language. S creates a new object, in effect a context, that encapsulates the information of the source, and populates the object with results of an aggregation operation over the source information. The application that motivates the existence of the

S operator is data classification. The aggregation over the source data effectively groups the source into equivalence classes. Given contexts c_1, c_2, a ruleset e that defines the coherence expression, the syntax of the operator is as follows:

$$c_2 = \mathsf{S}_e(c_1)$$

Formally, the matching predicate e partitions the objects of the initial context c_1 into n equivalence classes. The constructed result context c_2, is an object consisting of $n+1$ values: the first is e, and the following n values are sets $s_1 \ldots s_n$ of references to each of the objects of c_1. One of the equivalence classes of the result context is an exception class, for objects that do not match e.

Since it is difficult to follow the capabilities of the S operator from a description alone, we present an extended example from our research.

3.3 Repository Refinement, Maintenance and Reuse

In this section we see how we use the algebra to refine the nexus iteratively by discovering anomalies in the source data and incorporating any resulting fixes into the context of the articulation. The new rules that we add after the analysis of the uncovered anomalies contribute to the consistency guarantees of the context. We show how this process is equivalent to the maintenance process when the underlying source data changes.

3.3.1 Iterative Context Refinement

This section shows how the exception classes of one iteration of an operator are expressed in new rules that extend the articulation for another iteration of the operator.

The S operator provides a simple method for assessing the contents of a context. For example, we use a simple AMORL rule together with S to split the dictionary head words into equivalence classes. The expression $\mathsf{S}_{\texttt{len(hw)div20}}(\texttt{dictionary})$ returns the entries of the dictionary grouped by length of the head words. Applying this operation on the actual data reveals terms with missing end tags in the data (implied by long head word length). Less than ten of these cases occur in the source data, and we easily construct individual rules to correct the run-on head words. Once the errors are identified, the rules to convert terms with missing end tags are added to the definition of the set '`hw`' above. Using S we are also able to determine that other tags were equally valuable as head words, that we need to remove accentuation from foreign words. We also discover spelling errors in the head words by analyzing the frequency of words found in definitions, but do not occur as head words. The context refinement algorithm is as follows:

1. For i in $\{1, \cdots, n\}$
2. $c_{i'} = \mathsf{S}_{e_i}(c_i)$
3. Generate rule(s) r_i to handle exception objects
4. Insert r_i into ruleset for c_i creating c_{i+1}
5. Generate e_{i+1}

3.3.2 Maintaining the Repository

This section shows how updates to the information sources requires adjustments to the articulation built on top of them. These adjustments are shown to be equivalent to articulation improvements defined above.

In the course of developing the wrapper to the Webster's dictionary the maintainer of the source data performed a major revision of the source, affecting 10%-25% of all of the entries. These changes are part of an ongoing effort to correct and extend the dictionary, and they included corrections in the tagging of the entries, spelling corrections, reformatting of the text, addition of notes and comments, etc. By maintaining statistics with the S operator on the process of extracting the relevant parts of the dictionary, we were able to note which rules were no longer needed because the exception they handled had been updated. A comparison of the terms that we could not classify in the old and updated sources revealed a few new errors that had been introduced in the data. Since the editorial effort for this on-line Webster's dictionary is volunteer driven, it is possible that the level of rigor in the updates is lower than for a professional publication. As it turns out there was relatively little within the wrapper that required correction when the source changed. In fact, only approximately 40 rules, or 8% of the total, were rendered unnecessary as a result of the source change. There were very few changes to the size statistics of the dictionary, and the wrapper execution time decreased slightly as a result of the reduced number of rules. Formulating the coherence measure within the algebraic framework significantly simplified the process of identifying and handling the changes.

3.3.3 Revisions for Reuse

This section shows that the operator code developed for one dictionary source is portable and scalable. We obtained source data for a 1911 edition Roget's thesaurus, which we processed using the Webster's dictionary articulation. The changes to account for the different source structure took approximately one half hour to implement. The compact resulting nexus with 1022 nodes and 5,100 arcs served as a convenient test bed for early experiments with the all pairs similarity algorithm described in Section 4.4.3. The success of this refactorization of the nexus articulation allowed us to characterize it better. The nexus articulation is easily applied to any source which refers to its own component parts, such as dictionaries, glossaries, thesauri, and encyclopedias. To confirm the generality of the articulation it was still important to show its scalability.

By adapting the code developed for the Webster's dictionary to operate on the Oxford English Dictionary, we tackled a data source that was over eleven times as large as the first. While the original wrapper was developed over a four month period, the code update required five weeks and 1000 lines of code, while the execution time of the refactoring code scaled linearly with the data size. The main effort involved segmenting the OED into chunks containing complete definitions. As with the Webster's dictionary, we found many cases where the tagging that was provided with the dictionary was insufficient to be used without modification. Specifically in the case of the OED there were literally thousands of definitional words that were found outside of the definition tags.

The resulting OED nexus is four times the size of the Webster's nexus. The articulation adaptation is also represented as an iterative improvement. Table 3.3 shows the ratio of sizes of the two repositories. The two entries in bold are the most significant in the scalability of the algorithms. The first is relevant in the initial construction of the repository, the second in the algorithms computed over the repository structure. Note that there is some duplication of arcs between nodes, since some words are repeated within many dictionary definitions. The unique arcs row in the table counts no duplicate arcs.

The major insight gained by this effort is, again, that whenever refactoring bodies of information for a new purpose, it is crucial to evaluate that the requirements of the new application are met by the semantics of the source. The initial "naive" expression to adjust the refactoring code for the Oxford English Dictionary was developed in a few days. However, this transformation uncovered literally thousands (7%) of the definitions empty or containing single word connectors such as 'Hence,' 'So' and 'Also,' and thousands

Table 3.3: Nexus Size Comparison

Repository Component	OED Size	Webster Size	Ratio
raw data	570 MB	50 MB	11.4
terms	510,984	112,897	**4.53**
nodes	327,680	96,800	3.39
arcs	8,090,735	2,021,549	**4.00**
unique arcs	5,253,332	1,438,532	3.65

more containing the words 'see prev.' or 'see next.' An additional two weeks of coding was necessary to account for these factors. In contrast to the four months of effort to create the Webster's repository the OED repository, building on the existing code base only required a total of five additional weeks. Although four times as large, the OED repository needed one quarter the development time. This result indicates that our framework is an efficient one for reuse as well.

It is important to note that the source semantics of the OED are not incorrect. In the actual text of the dictionary, a definition such as 'Hence', is always followed with a more complete definition, for a subsequent equivalent term. Likewise, it is easy to visually refer to a previous definition, when the current term's definition reads 'see prev.' However, in their existing form, these semantic references to other definitions were unsuited to the novel application which we were targeting. Our solution was to segment the OED in such a way that these terms with referential definitions were always grouped together with the previous, or where appropriate, with the following term.

Table 3.4: Repository Construction Times

Construction Phase	OED Time	Webster Time	Ratio
segmentation/glossarization	72 min.	14 min.	5.14
graph matching	134 min.	21 min.	6.38
ranking functions	309 min.	77 min.	4.01
overall times	9.5 hr.	1.85 hr.	4.60

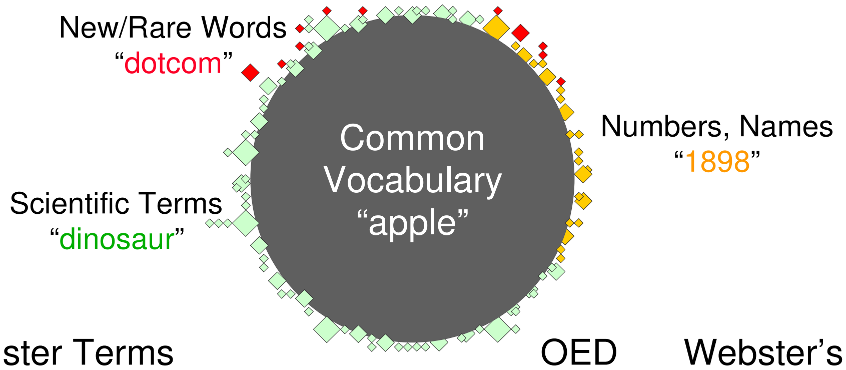
Table 3.4 gives the ratio in execution time for the construction and ranking of the two repositories. The first two phases have a computational cost that is dominated by the terms of the repositories. Effectively, the first phase finds all of the repository terms, and the second phase matches up definition words to those terms. The second order cost in the first two phases is the size of the raw data, since both phases must pass through the entire data set to find the terms and definition words. The ranking phase is dominated

in computational cost by the number of arcs in the repository graph, since a function is repeatedly computed for each arc. Overall, we observe a linear growth in computation time with respect to the size of the source data.

In order to check that all of the rule transformations execute consistently, we developed visualization techniques in parallel with the articulation. Also, the process of developing algorithms to exploit the nexus proceeded through trial and error. The first step we took towards these algorithms was to explore the structure of the nexus. In the next section we discuss the first tools developed to visualize the repository structure.

3.4 Graph Manipulation Toolkit

This section presents tools for graph exploration, and visualization, that expose the overall structure of labeled directed graphs. Prior to this work, there was no description of the overall structure of the graph of definition terms of the dictionary. The tools below exposed this structure, and made it possible to determine what kinds of graph algorithms would be well suited for such graphs. One discovery due to these tools was the tight coherence of meaning for the terms in the thousands of word clusters found in the nexus, such as the one shown for the term 'Egoism' in Figure 3.2.



Cluster Terms	OED	Webster's
Strongly Connected Component: Kernel	82692	27005
Connect into Kernel	244988	69795
Connect from Kernel	27442	3931
Disconnected	15709	4938

Figure 3.3: Overall Dictionary Nexus Structure

The word nexus contains a central cluster or *kernel*, which is strongly connected, that is, there exists a cycle of directed arcs between any pair of nodes. The nexus is surrounded by a cloud of much smaller clusters, which link to the kernel, but no nodes link to them. After processing by the graph manipulation tools, the kernel and satellite clusters emerge from the tangle of repository links. Figure 3.3 presents this general structure and lists the total number of nodes in the various types of clusters. The figure depicts the great difference in relative size between the different types of cluster, but only very roughly approximates the relative numbers of the cluster types. The central circle represents the kernel, which contains common vocabulary of the English language, such as the word '**apple**.' Words like '**dinosaur**' which are not used as frequently, form clusters that connect into the kernel, while dates, '**1898**' and proper nouns '**England**', which are not defined in the dictionary, but are used in the dictionary definitions form clusters that are connected from the kernel. In every living language there exist terms which have not entered the dictionary, because they are new or have limited usage, like '**dotcom**'. For completeness we have included these terms in Figure 3.3 in the disconnected category. We have approximated their number by considering in this category all terms that appear in exactly one definition, but aren't defined themselves. Other estimates of the number of disconnected terms could be obtained by computing the number of terms used in secondary sources that do not appear in the dictionary. We do not consider the disconnected category any further for lack of data. Figure 3.3 also gives the number of terms that fall into these categories in the OED nexus and the Webster's nexus.

Given this overall graph structure consisting of one large component surrounded by a multitude of small components, it is interesting to examine the distribution of incoming and outgoing arcs for each node. These distributions should provide clues as to why the graph is structured as it is. Figure 3.4 shows that for both the Webster's and OED repository, the in-degree of the nodes of the graph follows a Zipfian [Zip49] distribution with the same exponent. Likewise, Figure 3.5 shows the out-degree of nodes in both repositories is Zipfian, except for definitions under twenty words in length, which represent 20% of the arcs of the graph. It is striking that these distributions arise in independent repositories that represent the same entity, namely the English language. The distributions, and therefore the macroscopic structure of the graph, must arise out of properties of the language.

The tools described in the next sections were developed to explore the structure of the large directed graphs that form the nexus. These tools allow us to empirically verify that the nexus conforms semantically to the source dictionary from which it is extracted. They represent the early efforts at making use of the nexus before the algorithms of Chapter 4

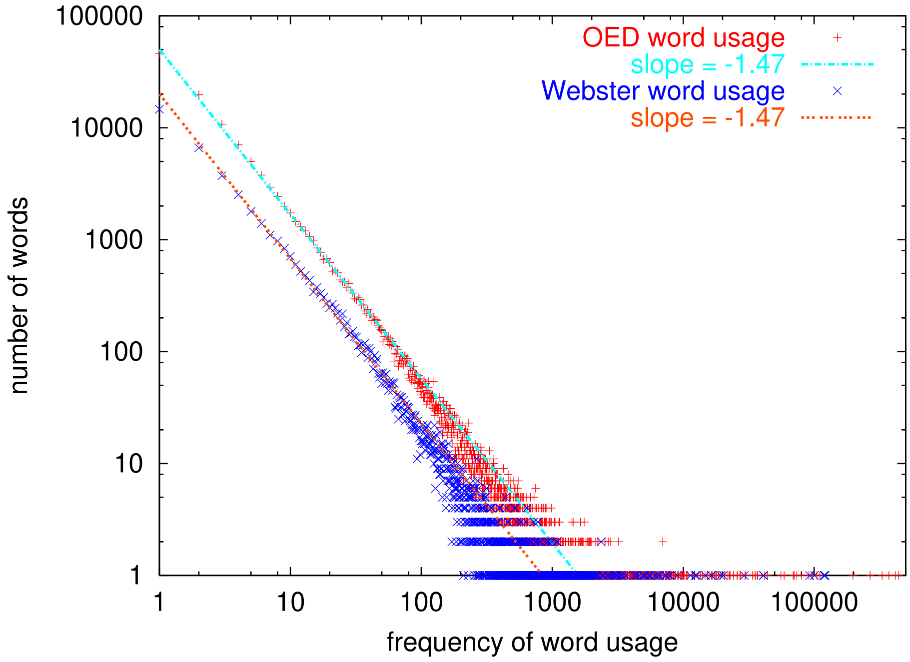


Figure 3.4: Distribution of Incoming Arcs

were invented.

3.4.1 Peeler

The graph peeler is a tool that iteratively cuts nodes from a directed graph, when they have no incoming links. As an example, a singly-linked list loses its first element to the peeler at every iteration, until it is an empty list. In contrast, a cycle of links never loses an element. Simple garbage collectors use this reference counting mechanism to reclaim memory space no longer needed by an application. This algorithm is linear in the number of arcs. The purpose for using the peeler is a divide and conquer strategy. The hope is to reduce the nexus to a size where its visualization and use is manageable.

3.4.2 Kernel Finder

The *kernel* finder takes a graph as input and returns the largest strongly connected component of the graph. A directed graph is strongly connected when there exists a round trip through the graph between any two nodes of the graph. The graph peeler serves as a preprocessing step to improve the efficiency of the process. We developed the kernel finder

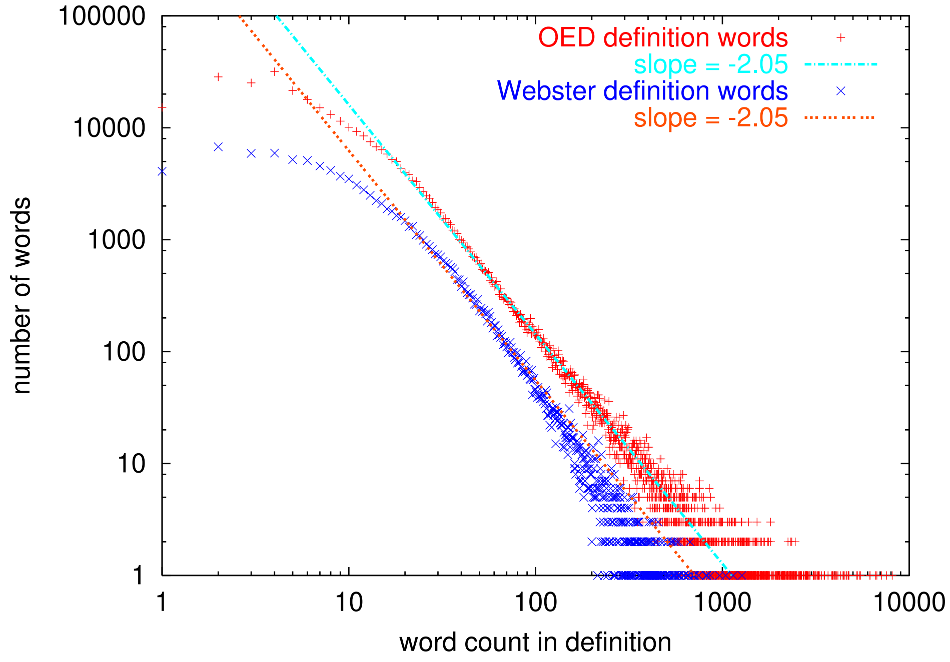


Figure 3.5: Distribution of Outgoing Arcs

once we discovered that the peeler reached a point where the nexus stop shrinking in size.

Once we have computed the kernel of the graph, we examine the clusters of nodes that are not in the kernel. These so-called *satellite* clusters are not required to be strongly connected. A cluster is defined as follows: each cluster has a seed that only has arcs to the kernel. Add nodes to the cluster if they have an arc to the seed, or to another node in the cluster. The seed is always a single node. If the seed is a pair of nodes that refer to each other, and to nodes in the kernel, we show by contradiction that this pair must also belong to the kernel (we assumed this is not the case).

3.4.3 Arc Filter

Now that we have defined the notion of cluster, it is interesting to visualize them. To do this, we first remove extraneous arcs. These are arcs from the cluster to the kernel. By removing them, we are left with the the arcs that define the cluster. The arc filter takes two graphs as input, and returns a subgraph of the first, with no arcs to nodes in the second.

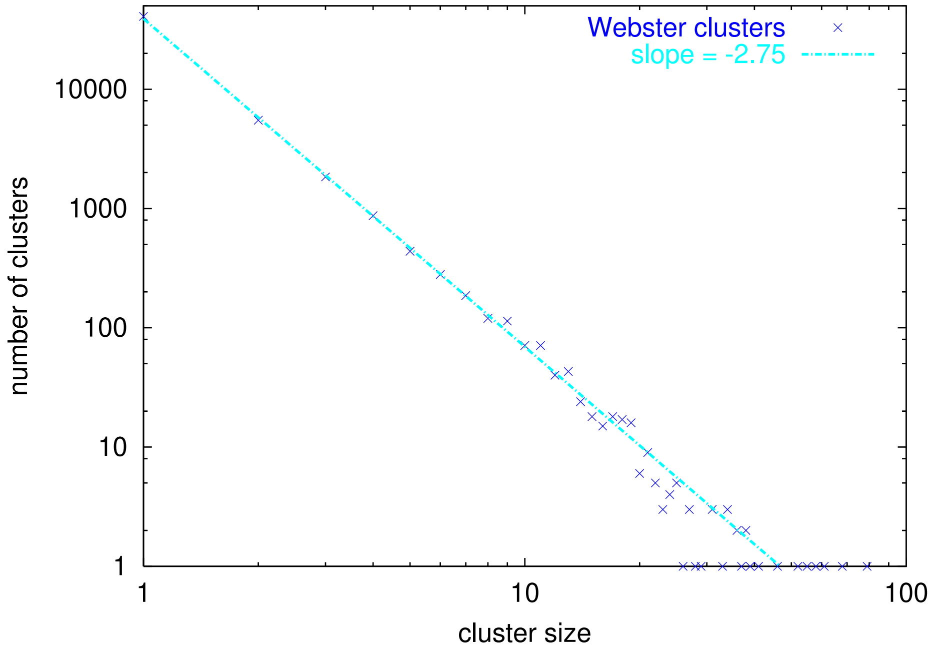


Figure 3.6: Webster's Nexus Cluster Size Graph

3.4.4 Cluster Joiner

We may join clusters together when they share nodes. This situation occurs when a node has arcs that refer to terms in otherwise distinct clusters. By joining clusters, we have the ability to consider possible meaning associated with the aggregated set of terms. As an example, one joined cluster contained sub-clusters relating to marine dinosaurs, dinosaurs of the Triassic age, flying dinosaurs, as well as a broad category of dinosaurs. This super-cluster has single nodes that join the sub-clusters, each to the broad category of dinosaurs. The value of joining the clusters is to be able to visualize the larger structure and recognize that it has its own unity of meaning as well. Figure 3.6 shows that the clusters, as well as the joined clusters, follow a Zipfian distribution according to their size. The satellite clusters are sorted in decreasing order of size before plotting the graph. We predict the likelihood of a cluster of any given size in the dictionary by constructing a probability distribution based on a power of the cluster size. The exponent is given by the slope of the line that approximates the plots in the figure. Figure 3.7 shows that the OED clusters are Zipfian with a different slope. Since the in-degree and out-degree statistics are so alike, as well as the ratio of the size of the kernel to the size of the nexus, it appears that clusters do

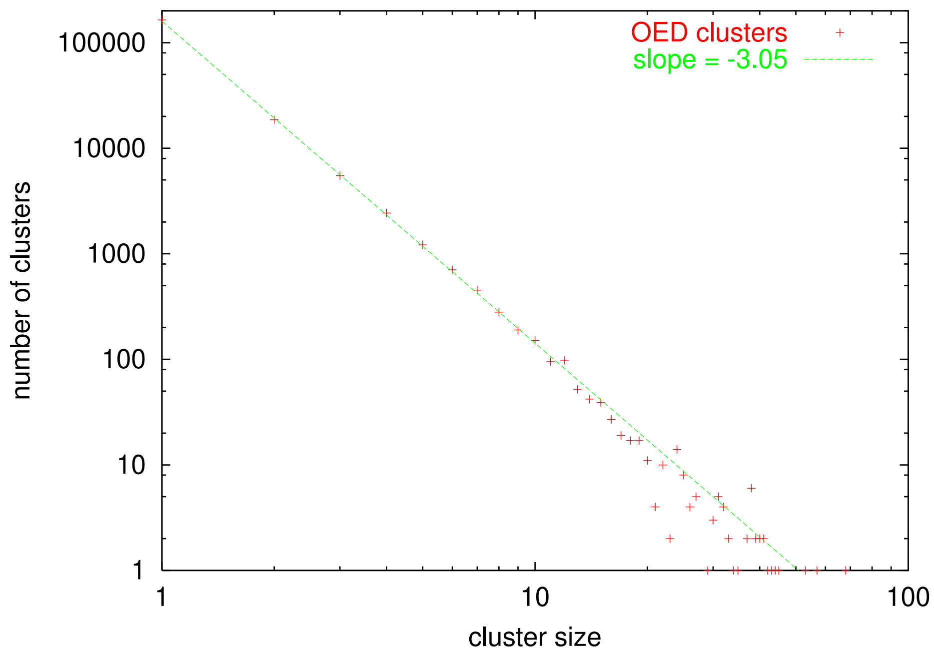


Figure 3.7: OED Nexus Cluster Size Graph

not grow beyond a certain size before being absorbed into the kernel. Table 3.5 gives more statistics on cluster sizes and their number. Note the small median size of satellite clusters. This statistic is an indication of how preponderant the importance of the kernel is.

By doing this analysis of the structure of the nexus we have been able to ascertain the consistency of the rules used to generate it. The clusters we extract from the nexus show that dictionary definitions define semantic neighborhoods, and that our articulation preserves them. The properties of the arcs in the nexus lead us to the intuition that both in-arcs and out-arcs are important to the nodes of the nexus. It is this realization that informed the development of the algorithms we present in Chapter 4.

3.4.5 Visualization Front End

This section describes two graph visualization techniques, one for small sparse multi-level clusters, the other for richly connected objects. The first centers longer object chains, the other ranks and centers objects around important incoming and outgoing arcs. Figure 3.8 illustrates how satellite clusters from the graph can be visualized. After applying the graph manipulation tools to the dictionary repository, the clusters smaller than 20 nodes in size are

Table 3.5: Cluster Count and Size Information

Type	OED	Webster's
total cluster count	189,149	50,485
singleton satellites	164,086	40,742
multi-node satellites	25,063	9,743
nexus size	327,680	96,800
kernel size	82,692	27,005
median satellite size	2	2
average satellite size	3.23	3.69

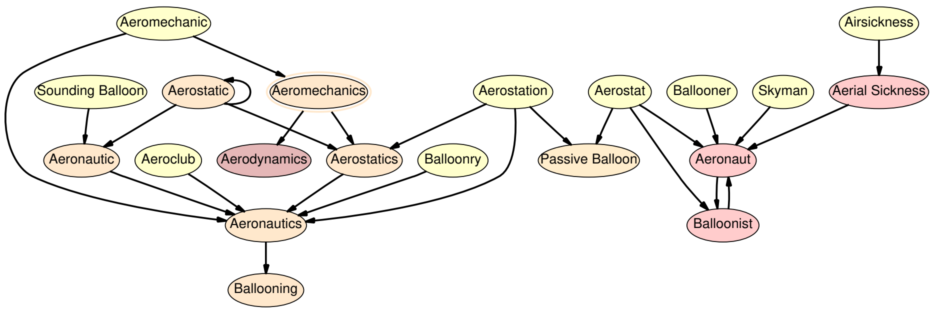


Figure 3.8: Sparse Cluster Visualization

easily rendered using graph visualization tools such as *dot* from AT&T Research [ATT99]. Figure 3.2 is another representative example of the first style of visualization, while Figures 4.4 on Page 84 and 4.11 on Page 91 are representatives of latter style. Note that the central node and other nodes of Figure 4.11 have many more arcs than are displayed. The actual number of outgoing and incoming arcs for the central node is marked in the central node. In order to select the best nodes to display, we need a mechanism to select the most significant arcs for any given node. This is the topic of Chapter 4.

3.5 Comparative Analysis of the Nexus

In this section we compare the macroscopic structure of the nexus to another large structure, the World Wide Web. We also compare the nexus to two related lexical systems, WordNet and MindNet.

3.5.1 Comparison to the Structure of the World Wide Web

The most striking aspect of the structure uncovered in the previous sections is that it is invariant across the two dictionary repositories. What's more, this structure is shared by the World Wide Web when it is viewed as a directed graph. Recent results of Rajagopalan [RBK+00], based on a survey of 200,000,000 web pages, give the Web a kernel of 56,000,000 pages. The in degree, out degree and cluster size distributions are all Zipfian with exponents 2.1, 2.7, and 2.54, respectively. Contrast this with exponents of 1.47, 2.05 and 2.75 (3.05) for the corresponding distributions in the nexus. We hypothesize that the difference in exponents stems from the much greater volatility of web pages as compared to words in the English language, but that both structures are shaped by the same type of evolutionary process. This is very strong evidence supporting the use, over the dictionary repositories, of algorithms that have proven value in the graph structure of the Web.

3.5.2 Comparison to Other Systems

This section compares the nexus to two other repositories, the first manually constructed, the second generated by phrase parsing a dictionary. We discuss where the dictionary repository approach expands on the capabilities of existing lexical repositories. We show the broadness of the repository as a complement to the preciseness of the relationships in other repositories.

WordNet has been in development since 1990, using a painstaking manual process, and its design has been elaborated since 1986. Its current revision, WordNet 1.6 was released in 1998, and includes four principal data files, and a number of programs to aid in searching and displaying the data. Of the existing electronic lexical tools, WordNet is the one that most closely resembles the Webster's repository. Recently Plumb Design [Plu00] has made a java applet visualization available of WordNet, in which each term connects to the others of the same category. This technology demonstration, is revealing. It gives a powerful visual indication of the sparseness of the relationships between terms in the WordNet system. In the visualization of the WordNet graph each term connects to a small number of other nodes, on average less than five. Choosing the simple metric of average number of arcs per node, Table 3.6 shows that the nexus has between two to three times as many of these relationships, despite its much larger number of nodes.

Table 3.6: Comparison of Repositories to MindNet and WordNet 1.6

Name	Location	Nodes (1,000s)	Senses (1,000s)	Arcs (1,000s)	Devel. (month)
Webster's Nexus	Stanford (Jannink)	97	113	1439	4
OED Nexus	Stanford (Jannink)	327	511	5253	1
WordNet (Miller)	Princeton	100	174	727	~96
MindNet (Richardson)	Microsoft Research		159	713	~48

MindNet is not publicly available, but its scale is 159,000 head words and 713,000 relationships between head words. Its development began in 1992, and it supports 24 different relationship types between terms. MindNet relies on phrase parsing and grammatical analysis to extract these different relationships. According to Richardson [Ric97], its initial developer, the *part-of* relationship was only correctly extracted 15% of the time. Dozens of specialized parsing structures had to be inferred in order to account for the non-grammatical phrases in a large percentage of the dictionary definitions. As a result, it would appear that MindNet suffers from problems, both in terms of accuracy and completeness of extraction.

The relationships WordNet defines between terms are more precise, as they were manually entered; however, there are necessarily fewer of them, and they are far from exhaustive. Also, since the design of WordNet long preceded its implementation, artificial concepts, such as non-existent words, and artificial categorizations, such as non-conforming adjectives, were introduced when the repository was built. These constructs are a valid ad hoc approach to make the terms conform to the design, but they do not arise out of the usage of the language. WordNet carefully distinguishes between senses of a term, and separates a term into multiple entries when it may be used as different parts of speech, i.e., to *run* vs. a computer *run* vs. a *run* salmon (a run salmon is a salmon that has completed the return to its spawning grounds). The Webster's nexus distinguishes senses of a term based only on usage, not on grammar. Another significant difference between the two structures is that the data in WordNet is separated by lexical categories, whereas our dictionary repositories allow any relationship between terms to exist. Table 3.6 makes some simple numerical comparisons between the two systems.

The most striking contrast in Table 3.6 is that the OED nexus contains almost seven times the number of unique relationships between terms as WordNet, but at a development cost that was orders of magnitude smaller. Using the OED makes it possible to cover the breadth of the English language, and capitalize on the decades of work that went into that opus. Having compared the repositories numerically, it is necessary to illustrate with an example what the Webster's repository provides. Specifically, it relates terms without defining the type of relationship, just the importance of the relationship. In Section 4.2.3 we examine some subgraphs that emerge from the repository data, based on terms relating to transportation.

3.6 Nexus Review

In this chapter we have examined the creation of two large scale repositories, the Webster's nexus and the OED nexus, explored its structure and discussed the procedure used in its construction. We have shown that the techniques used in the construction are usable in different settings, and demonstrated the scalability of the operators used in the nexus construction We also showed that these operators are efficiently maintainable when the source data changes. We verified empirically and statistically that the construction of the nexus was semantically consistent with the contents of the source dictionaries. The construction time for the OED nexus scaled linearly from the construction time for the Webster's nexus.

Chapter 4

Word Nexus Algorithms

Chapter Outline

This chapter describes the ArcRank model of relationships between nodes in a directed labeled graph, such as hypertext. ArcRank fulfills the second of our hypotheses in this dissertation, which holds that the nexus algorithms are scalable and make it possible to use the nexus efficiently to match information from diverse sources. We present a ranking algorithm for directed arcs, and an algorithm for extraction of hierarchical relationships between words in a dictionary. Using ArcRank we create a thesaurus style tool to aid in the integration of texts and databases whose content is similar but whose terms are different. These algorithms produce repositories that complement handcrafted thesauri by determining more complete relationships between words, although they are less specific. Exploiting hierarchies of relationships between words paves the way for broadening and related term queries in web-based repositories. In this chapter we also show how the ArcRank model of arc importance allows us to define similar paths, and compute all pairs of similar nodes. More precisely, when nodes are reachable through similar arc paths, or reach similar node sets, then they themselves are similar. The all pairs similarity algorithm is efficient on the dictionary nexus graphs, which have Zipfian arc distributions. Using ArcRank, we apply a simple generalization of the pattern/relationship extraction algorithm to generate sets of nodes having a *kinship* or near synonymy relationship.

4.0.1 General Intuition

The principal obstacle in integrating information from multiple sources is their semantic heterogeneity. The most easily recognized form of heterogeneity is when different terms are used to mean the same thing: lexical heterogeneity. Even more frequently terms have some incomplete overlap of meaning, and we can recognize the cases of overlap. Even so, there is no algorithmic procedure to resolve problems of lexical heterogeneity authoritatively. However, we still desire assistance in determining semantically related terms.

The starting point for this research is the hypothesis that structural relationships between terms are relevant to their meaning. These relationships become interesting when all items in the domain of interest contain the relationships, and are organized according to them. Dictionary definitions form a closed domain in the sense that the words used in definitions, taken as a set, should be defined elsewhere in the dictionary. This property leads to a directed labeled graph representation of the dictionary. Nodes of the graph model definitions, head words are labels for the nodes, and a word in a definition represents an arc to the node having that word as a label. Notable collections which are not closed include encyclopedias, which cover a selected partial set of terms equivalent to the dictionary nouns together with proper nouns, and search engines, which return documents for all but stop words. Thesauri are not closed either, since the categories which group terms together form a kind of meta language, different from the terms found within the categories. This is especially true for categories that contain verbs.

PageRank is an algorithm that ranks objects according to the ranking of objects that refer to them. At first glance, the PageRank model of Web structure [PB98] does not lend itself to direct application in non-hypertextual domains, since it relies on the link structure to compute a ranking over items. However, we have found that a related model, which we call ArcRank and define in Section 4.2, is useful for extracting relationships between words in a dictionary. This model expresses the importance of a word when used in the definition of another. The attraction of using the dictionary as a structuring tool is precisely that head words are distinguished terms for the definition text. This extra information allows types of analysis that are not currently performed in traditional data mining and IR, where no term is assigned as 'head word' of a document.

Using the extraction technique discussed in Chapter 3 we generate such a graph structure and then use it to create thesaurus entries for all words defined in the structure, including *stop words* such as 'the,' 'a,' 'of' that most systems specifically ignore. The thesaurus engine, based on our relationship ranking technique, constructs more complete repositories than manually constructed thesauri, since every term has at least one relationship defined through its dictionary definition. These relationships are less specific than those in the manually constructed thesauri. The thesaurus graph is a potentially important tool for systems integration experts. More interestingly, we also find that the algorithms that generate the thesaurus may be applied to document classification and the ranking of results of mining queries.

Given a directed graph whose nodes and arcs have been ranked according to the ArcRank model, it becomes possible to consider the extraction of hierarchical relationships between terms. Indeed, the structure made explicit in the data, as in Figure 4.9, indicates that it is rich enough to find terms similar to locomotive, based on the terms which relate to locomotive. We show how the all pairs similarity algorithm allows us to group terms by kinship, according to their link structure. We show how these clusters, along with the accompanying link structures, also define ancestor candidates for the clusters. The number of ancestor candidates vary according to the selection and scope of the kinship clusters. These algorithms are the tools that make the nexus an integral part of the SKEIN system.

4.1 Term Importance from Graph Structure

This section motivates the iterative measurement of an object's rank, based on its structural relationships in the object graph to which it belongs. We present the basis of our dictionary structuring techniques. Before presenting the ArcRank measure, we present the PageRank algorithm and the variants we have used in our experiments.

4.1.1 PageRank

The PageRank algorithm is the initial step of the ArcRank ranking technique described in Section 4.2, and is important to define before discussing the ranking of arcs. Table 4.1 shows a pseudocode description of the algorithm.

The PageRank algorithm is a flow algorithm over the graph's arcs. It assumes no capacity constraints limiting the amount of flow possible between two nodes. All nodes begin with an initial ranking, in our case a constant $1/|n|$, where $|n|$ is the number of nodes in the graph. At each iteration, nodes distribute their rank to their neighbors on outgoing arcs, and receive rank from neighbors on incoming arcs. The total outgoing flow from a

Table 4.1: PageRank

input: directed graph, *output*: scored node list

1. Make adjacency list representation of directed graph
2. Make rank array of size $|n|$ for graph nodes
3. Set (round 0) rank $p_{0s} = 1/n$ for all nodes s
4. While `rankchange` $>$ `threshold` (round i)
5. For nodes s in $\{1 \cdots |n|\}$ (ranking step)
6. For arcs $a_{s,t}$ in s's adjacency list a_s
7. Transfer rank $p_{is}/|a_s|$ from source s to target t
8. For nodes s in $\{1 \cdots |n|\}$ (adjustment step)
9. Normalize, if needed, rank p_{is} wrt to total rank
10. Compute `rankchange` from previous iteration
11. Return final values from rank array

node is never greater than its rank, $\sum_t a_{s,t} \leq p_s$, nor is any individual $a_{s,t}$ ever less than zero. The intuition behind the flow is that more richly connected areas of the graph carry larger capacity, and therefore nodes in these areas maintain a higher rank. The rank flow of nodes in strongly connected aperiodic graphs is shown to converge to a steady state [MP95]. Steady state flow is desirable, because it allows us to assert stable relationships between nodes in the graph. In practice, we accept variability in the flow between nodes, as long as the total variability over the entire graph lies below a threshold.

In general graphs, nodes and clusters of nodes with only outgoing arcs act as sources which lose all of their rank. Likewise, nodes with incoming arcs only act as sinks for the rank of their neighbors. The dictionary graph contains both source and sink nodes: source nodes represent words which are never used in other words' definitions, sink nodes are words whose definitions are not found in the dictionary. Figure 4.1 shows a representative subgraph of the dictionary repository centered around the node Fruit. In this subgraph, Marmalade is a source node, while Fleshy is a sink node. In the repository taken as a whole, sinks consist of misspellings, proper nouns such as geographical and Latin species names, and scientific formulae, which we do not consider. In PageRank, the rank of sources, sinks and weakly connected clusters do not reflect their local structural differences well. The intuitive idea is that sources and weakly connected clusters lose all of their rank to the overall graph, while sinks continuously accumulate rank, giving them an increasingly disproportionate

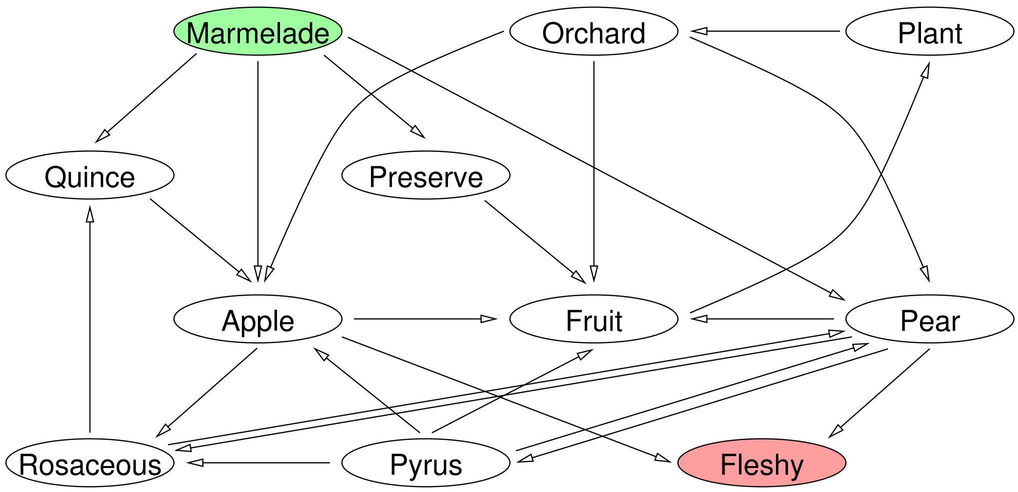


Figure 4.1: Source and Sink Nodes in Dictionary Subgraph

importance.

In our algorithm, the final rank of a node should be defined in such a way that, when any two nodes have a distinct pattern of connections, then their rank will differ. We adapted the algorithm from Table 4.1 in each of the following four ways so that sources and weakly connected clusters preserve some rank at each iteration. Ultimately we settled on the fourth technique, as it presented a significant performance benefit.

1. redistribute $b\%(b/100)$ of total graph rank before each iteration
2. limit rank transfer to a fraction $1/c$ of a node's rank
3. add a self-arc $a_{t,t}$ (node t is both source and target) to nodes
4. add a *gateway* node g and arcs $a_{g,n}$ and $a_{n,g}$ for all nodes n

It is sufficient to set a non-zero threshold for termination of PageRank, and to use one of the above adaptations, to ensure that all graph nodes preserve a non zero rank. Figure 4.2 shows a small subgraph of the repository centered around Fruit, with the addition of a gateway node. The arrows from the gateway node are bidirectional to indicate they consist of an arc in each direction, as defined above. We show here that, given a node t, at iteration i with rank p_{it}, the following holds:

Theorem 1 $\boxed{\forall t \in G, p_{it} > 0}$

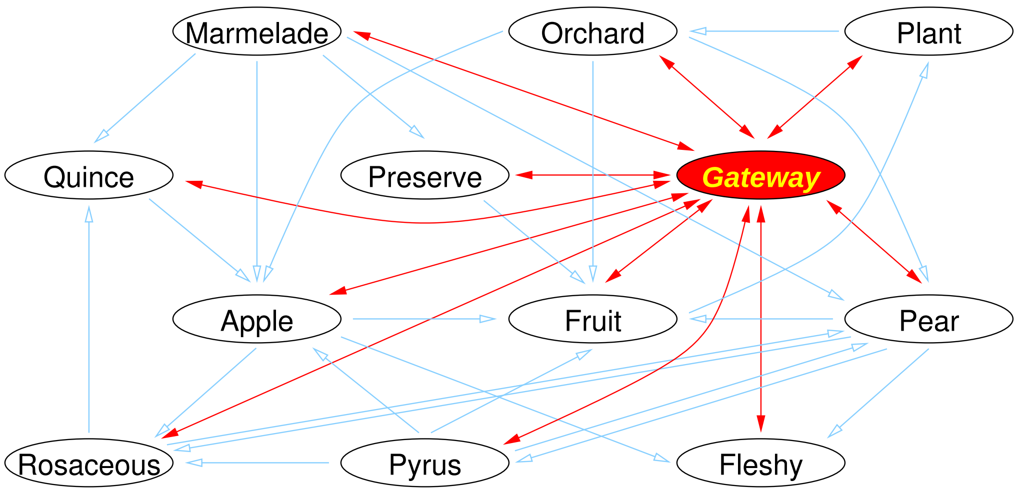


Figure 4.2: Addition of Gateway Node to Dictionary Subgraph

Proceeding by induction, we have: by definition, at the initial iteration, $p_{0t} = 1/n > 0$. *Assuming the property holds at iteration i, the following holds:*

$$p_{i+1t} = \{b/100, 1/c, p_{it}/(|a_t| + 1)\} + \sum_{v \neq t} a_{v,t}$$

Since, by definition, all quantities on the right hand side are positive and greater than zero, p_{i+1t} *is greater than zero. As indicated by the equation, this property holds for each PageRank variant enumerated above.*

We see that PageRank for dictionary terms represents the transitive contribution of each term to the definitions of all of the dictionary terms. We capitalize on this property to compute the relative importance of terms with respect to each other. This measure is a feature of the arcs between nodes, or equivalently in the dictionary, the *usage* of terms in the definitions of others. For the purpose of our discussion we define usage to mean occurrence in the dictionary.

Gateway Extension to PageRank

The gateway technique is defined above as a node g and arcs $a_{g,n}$ and $a_{n,g}$ for all nodes n of a graph. The gateway is a novel adaptation of PageRank and it provides a substantial improvement in algorithm runtime. Figure 4.3 shows the performance benefit of using the gateway. The convergence of the graph rank to a stable value is orders of magnitude faster

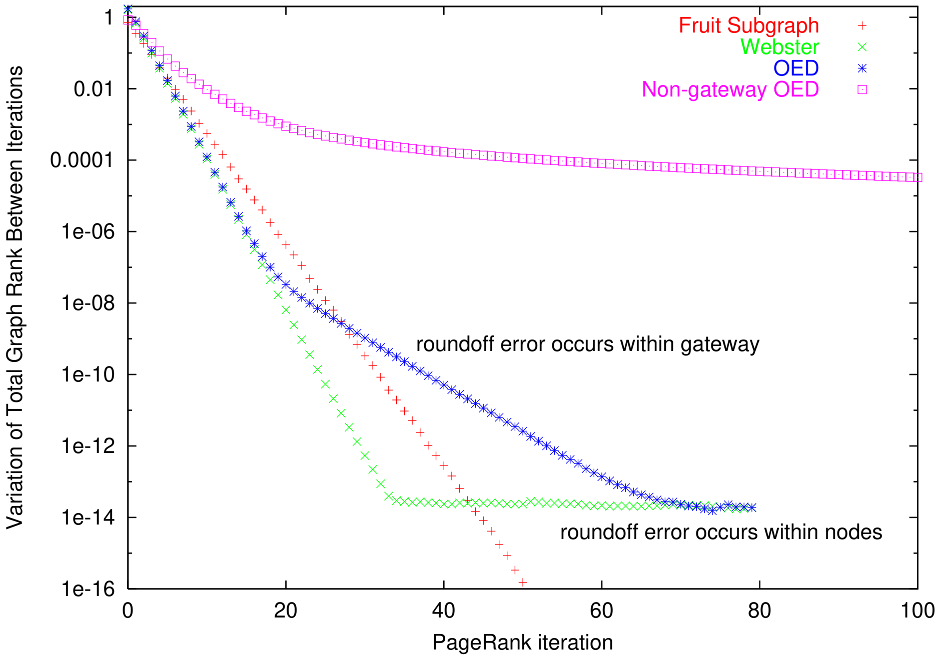


Figure 4.3: Convergence of Gateway Ranking Scheme

than the redistribution version of the PageRank algorithm, depicted by the light squares in the figure. An additional benefit of the gateway scheme is that it allows us to use the PageRank algorithm for all of the nodes that do not connect into the kernel. Previously, we had to discard all undefined terms such as dates from consideration in the nexus. With the gateway node it becomes possible to include them in the structure, and use them in the analysis of relationships between terms. The gateway technique preserves the overall structure of the graph, because it affects every other node of the graph in a minimal and isomorphic fashion. The gateway also guarantees convergence because the graph becomes a single strongly connected component. We finish by mentioning one feature of the graph, namely the angles in the OED and Webster curves. We have determined that these are cumulative roundoff errors in the gateway version of the PageRank algorithm. Fortunately the level of error, less than 10^{-8}, even accumulated over 60 iterations is well below the threshold of difference between average nodes in the graph.

4.1.2 Relative Arc Importance

In the dictionary application, PageRank suffers from some inherent limitations. First of all, PageRank is inherently a node-oriented algorithm. The top ranked nodes are the common conjunctions and prepositions, which convey little conceptual meaning, and are commonly considered stop words by other applications. It is clear that on its own, PageRank is insufficient to conceptually organize the dictionary structure. We may consider an extension to PageRank which assigns to each arc the amount of rank that flows across it at each iteration. As an absolute measure, this extension is also unsatisfactory, because it favors flows between the most highly ranked terms, that is, between stop words. Besides this obvious extension, there appears to be no self-evident technique to extract an absolute arc-based measure from PageRank.

However, our original goal is to identify the most important arcs for a given individual node. By casting our ranking problem in terms of our original goal we see that rather than an absolute measure, a relative measure between nodes is preferable. For any term in the dictionary, the words that signify the most in their definition should correspond to the arcs in the graph which are most significant in a ranking of arcs. Hence we arrive at the relative measure of arc relevance. Given an edge e, having source node s with rank p_s, target node t with rank p_t, and given $|a_s|$ outgoing arcs from s, the arc relevance r for e is defined as:

$$r_e = \frac{p_s/|a_s|}{p_t}$$

When s and t share several (m) edges $e_1 \ldots e_m$, we sum the arc ranks to compute the importance of t in the definition of s:

$$r_{s,t} = \sum_{e=1}^{m} \frac{p_s/|a_s|}{p_t}$$

$r_{s,t}$ measures the relative contribution of the rank of s to the rank of t which we show has desirable properties, such as:

Theorem 2 $\boxed{0 < r_{s,t} \leq 1}$

This follows directly from Theorem 1 and the definition of p_t*, since both numerator and denominator must be positive and* $p_t = \sum_v p_v/|a_v| = p_s/|a_s| + \sum_{v \neq s} p_v/|a_v| \Rightarrow p_t \geq p_s/|a_s|$.

Note that the arc importance measure is an indicator valid only in the immediate local vicinity of the end points of the arc. There is no reason to expect it to be globally commensurate. Having established an arc importance measure we are ready to present the ArcRank algorithm and walk through a hierarchical set of relationships the algorithm uncovers.

4.2 Arc Importance from PageRank

This section shows that arc importance is a local measure of flow in the stationary distribution of a directed graph. Arc importance provides a value that, while local to the arcs' source and target nodes respects a flow measure that is germane to the entire graph structure. Removal of low ranking arcs minimizes change to stationary distribution.

In the previous section, we computed a relative measure of arc importance. Here we show how to rank it with respect to both the source node and the target node, to promote arcs which are important to both endpoints. We discuss the repository we construct using ArcRank, and compare it to other systems.

4.2.1 ArcRank Algorithm Overview

Table 4.2: ArcRank

input: triples (source s, target t, arc importance $v_{s,t}$)

1. given source s and target t nodes
2. at s, let $r_s(v_{s,t_j}) = \text{Rank\_Arcs}(v_{s,t_j})$
3. at t, let $r_t(v_{s_i,t}) = \text{Rank\_Arcs}(v_{s_i,t})$
4. compute ArcRank: $a_{s,t} = mean(r_s(v_{s,t}), r_t(v_{s,t}))$
5. Subroutine Rank_Arcs *input*: list of arc importance values
   (a) sort arc importance values
   sample values $\{0.9, 0.75, 0.75, 0.75, 0.6, 0.5, \ldots, 0.1\}$
   (b) replace importance values with rank
   equal values take same rank $\{1, \mathbf{2}, \mathbf{2}, \mathbf{2}, \ldots\}$
   number ranks consecutively $\{1, 2, 2, 2, \mathbf{3}, \ldots\}$
   (c) return list of ranks

The ranking of an arc according to the arc importance metric defined above is typically different at the source and the target node. Indeed, it is possible for the highest arc importance value of arcs from a source node to be the lowest value for arcs coming into the target node. ArcRank, defined in Table 4.2, computes a mean of the ranked importance of arcs, so as to promote arcs which are important both to the source nodes and to the target nodes.

The form of the sample output of step 5 above $\{1, 2, 2, 2, \mathbf{3}, \ldots\}$ appears to be the only one that does not favor any particular type of node. This equal treatment of nodes is independent of the number of arcs and ranks associated with the nodes. Other rank numbering techniques resulted in skewed output. Competition style ranking, which counts equal values equally, but orders subsequent values differently, disadvantages arcs to nodes with many in-arcs. Given the same sample values from the above, the boldface value in the list here shows where competition ranking differs: $\{1, 2, 2, 2, \mathbf{5}, 6, \ldots\}$. Also, computing rank as a fraction of the total number of ranks: $\{1/n, 2/n, \ldots, \mathbf{n/n}\}$ favors arcs to nodes with a larger number of distinct ranks.

The ArcRank algorithm is more space intensive than PageRank, because it is arc oriented, but it is fast and easily made into a disk-based version. It essentially requires two passes through the data, and storage for twice the number of arcs. In the course of developing ArcRank, we derived a further extension to PageRank. The idea is to vary according to the arc importance ratio the amount of a source node's rank transfered to the targets. Tuning this optimization properly strengthens strong relationships and weakens less important ones. The additional cost of the technique is minimal, and it requires ranking arcs and summing ranks per node, before pushing value across arcs.

4.2.2 The Webster's Nexus

The repository we have built [Jan99b] has a very general structure, and it is defined by usage, that is, dictionary occurrence alone. There are no pre-imposed limitations, based on grammatical models, as to how terms relate. There is no attempt to perform a complete parse of the sentences in the definitions. This decision is necessitated by two concerns: the first is that dictionary definitions are often terse, non-grammatical sentences in which the parts of speech can be hard to identify. The second is that no parser exists that always correctly identifies the scope of every part of speech in a phrase. Since one goal of the structure is to use every definition word possible, without exceptions, it is not acceptable to use a parsing technique to preassign meanings to the graph arcs. Similarly, the scope of negation is not considered. As a result we often find opposites clustered together in the graph. Such clustering is appropriate for our objectives, since opposites are used very similarly.

This repository is the only one that does not exclude stop words, and as a result we are able to find that stop words most strongly relate to each other. On the down side, the type of relationship expressed in the repository is not always self evident, especially since many definitions and terms are now obsolete. Also, the accuracy of the ArcRank measure increases with the amount of data, and much of the dictionary contains very sparse definitions. Due to this sparseness we often find that the ArcRank scores of arcs to nodes with low PageRank is artificially high. The sparseness of data can also be misleading. A simple metric of ranking sources by the paucity of arcs works reasonably well, as long as there are relatively few arcs. In the general case, however, this metric does not hold up.

4.2.3 Browsing the Nexus

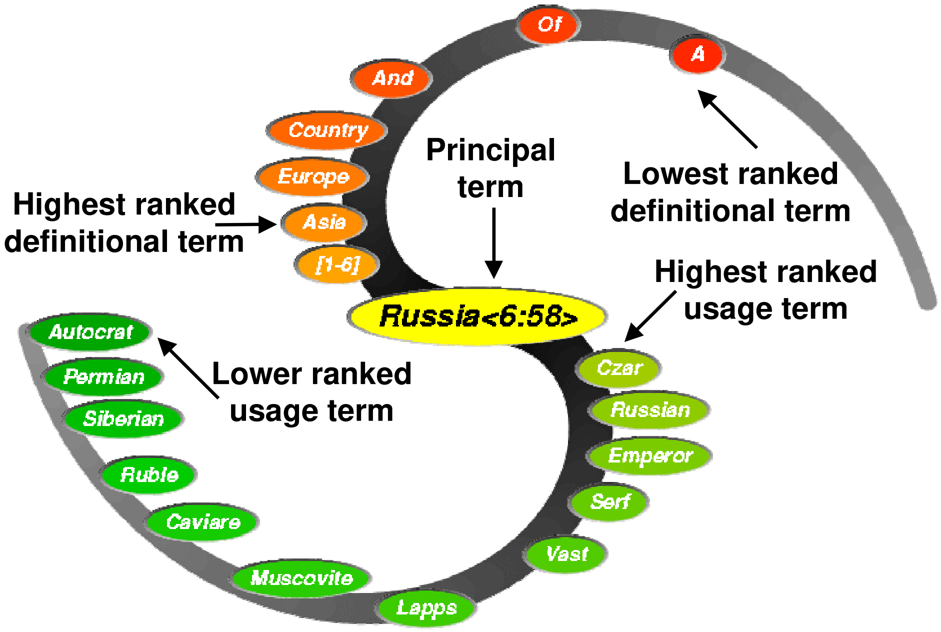


Figure 4.4: Sample Node from Webster Nexus Interface

Because of the size of the dictionary repositories, there is an inherent tradeoff when considering visualization techniques. The current trend in information visualization is to pursue sophisticated algorithms for displaying as large a portion of the graph structure as possible. This decision is often to the detriment of the legibility of the information in the nodes. Given our assertion that ArcRank is a high quality ranking of each node's incoming and outgoing arcs, we have chosen a different approach. Instead of displaying a hard to understand tangle of nodes and arcs, we have chosen for our browser as simple a display as possible. However, it still effectively communicates the notion of ranked relationships between nodes. The result of this effort is shown in Figure 4.4, and can be tested on-line at: `http://skeptic.stanford.edu/data/` [Jan99b, Jan00].

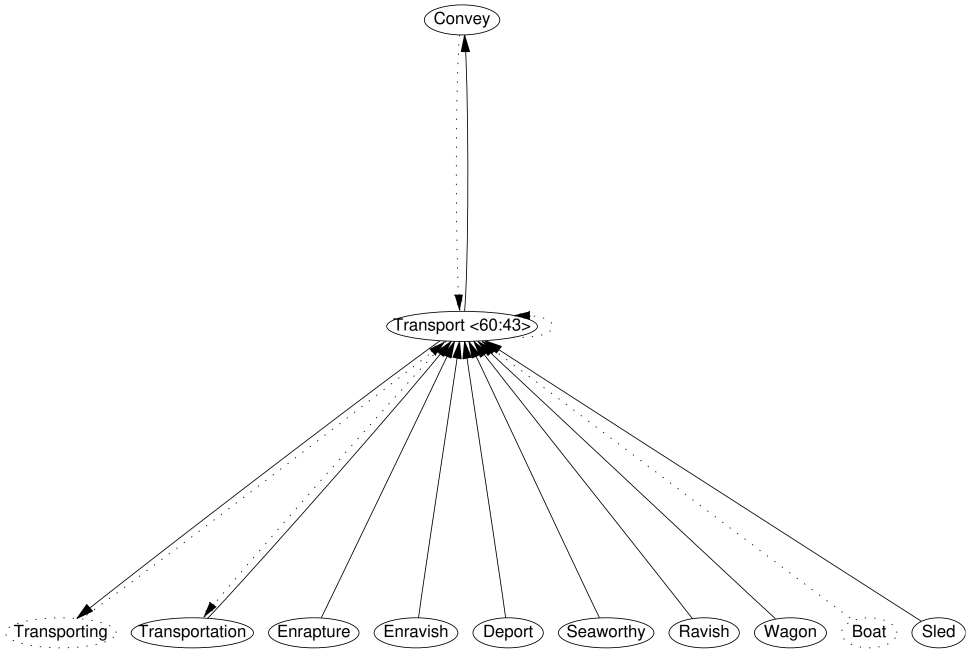


Figure 4.5: Terms Relating to Transport

We do not show all arcs for every node, but instead display the nodes associated with the highest ranked arcs. These are arrayed over a spiral curve, such that the nodes closest to the center are most closely related to the center node, based on the ArcRank measure. In Figure 4.4, with a little effort, we can deduce that the dictionary entry for Russia reads 'A country of Europe and Asia.' We find those words on the upper arc of the spiral, the *definitional* arc, which contain the words that are in the definition of the central or *principal* term. The words in the lower arc of the spiral, the *usage* arc, are some of the words that use the principal term in their definition.

Note that ArcRank ensures that the stop words And, Of and A have lower ranking than the other terms in the definition. In the set of terms that use Russia in their definition, the quality of the ordering is less immediately clear, but there is no question of the relatedness of these terms to the central node. In the discussion that follows, we use an interface that allows us to ascertain the cyclical case when a node is both in an other's definition, while using the other in its own definition (a situation that occurs more frequently than anticipated). Figure 4.5 shows arcs in both directions between nodes when the terms have each other in their definition. Transport is the principal node in this figure, and we see that it has arcs in both directions with Convey, Transporting and Transportation. Nodes in the bottom row of the graph that have a dashed oval have a PageRank value that is higher than that of the principal node. Nodes in the top row of the graph that have a dashed oval, such as in Figure 4.8, have a PageRank value that is lower than that of the principal node.

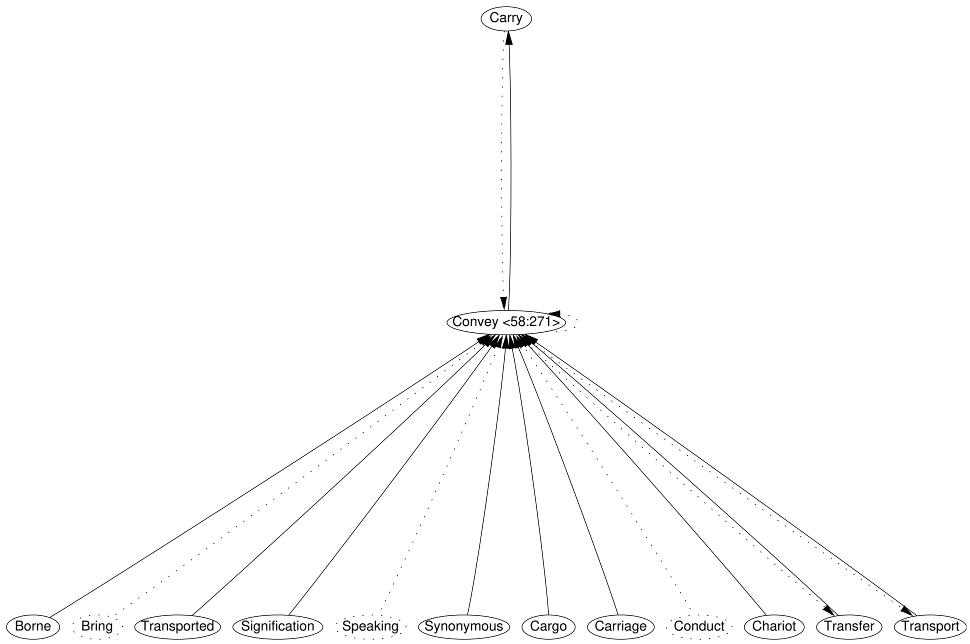


Figure 4.6: Convey Generalizes Transport

It is instructive to browse through the repository to get an idea of how it organizes the dictionary terms. The example below is prompted by an interest in developing a transportation ontology to support logistics applications. We start at the term Transport as shown below in Figure 4.5. The general form of charts generated using the repository, such as Figure 4.9, frame a term above by the terms used in its definition and below by terms that use it in their definition. These terms are placed from left to right in order of their ArcRank measure. The ArcRank values are not displayed in these figures to maximize the clarity and legibility of the charts. No more than the two dozen most significant associated terms are displayed: the label for the central term contains a count of incoming and outgoing arcs of the form <*outgoing, incoming*>. In addition to the ArcRank measure on arcs, each term has an associated PageRank value (not displayed). Arcs and Term borders are dotted when the arc's source node has a lower PageRank than its target node. We use this convention to recognize cases when ArcRank ranks an arc highly to a node that PageRank does not rank as highly.

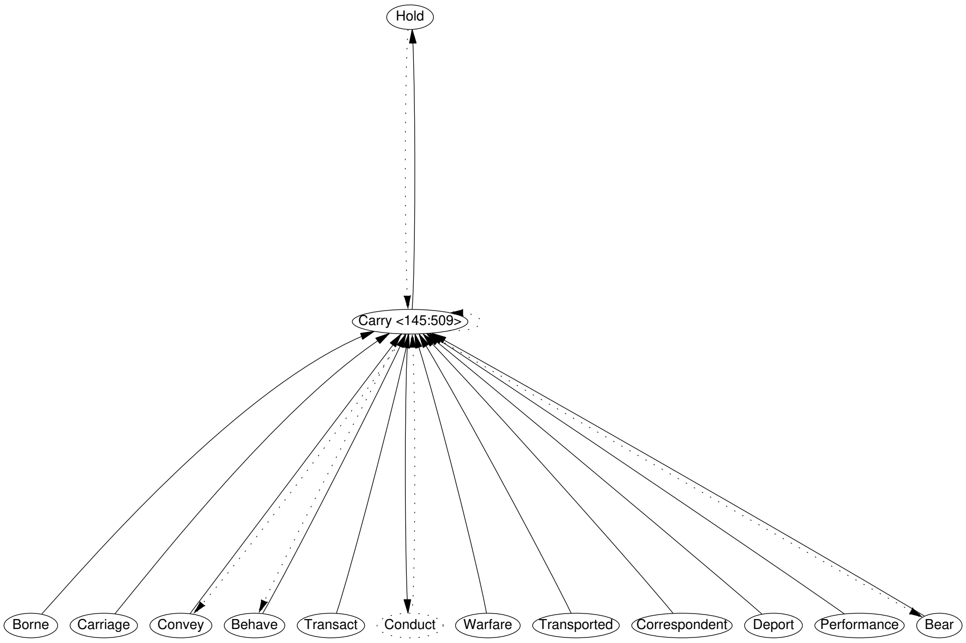


Figure 4.7: Carry Subsumes Convey

In Figure 4.5, which has been pruned for clarity, we see that the term Convey is used in transport's definition. Transport has multiple senses, including one relating to emotional transport, which is emphasized in the set of terms that use Transport in their definition. When we next examine the term chart for Convey, Figure 4.6, we find Transport, along with transported and cargo, which are also significant for the logistics ontology. Other terms in the set illustrate the more general nature of Convey as compared to Transport. Further browsing upwards in the repository takes us to the chart for Carry in Figure 4.7. Note how Carry subsumes Convey in the sense of transport, and that the term transported is also in its set of terms. We expect too that Hold, given its position as a parent of Carry in Figure 4.7, expresses a more general notion relating to Carry.

Starting from Transport in the downwards direction, we select Wagon and arrive at

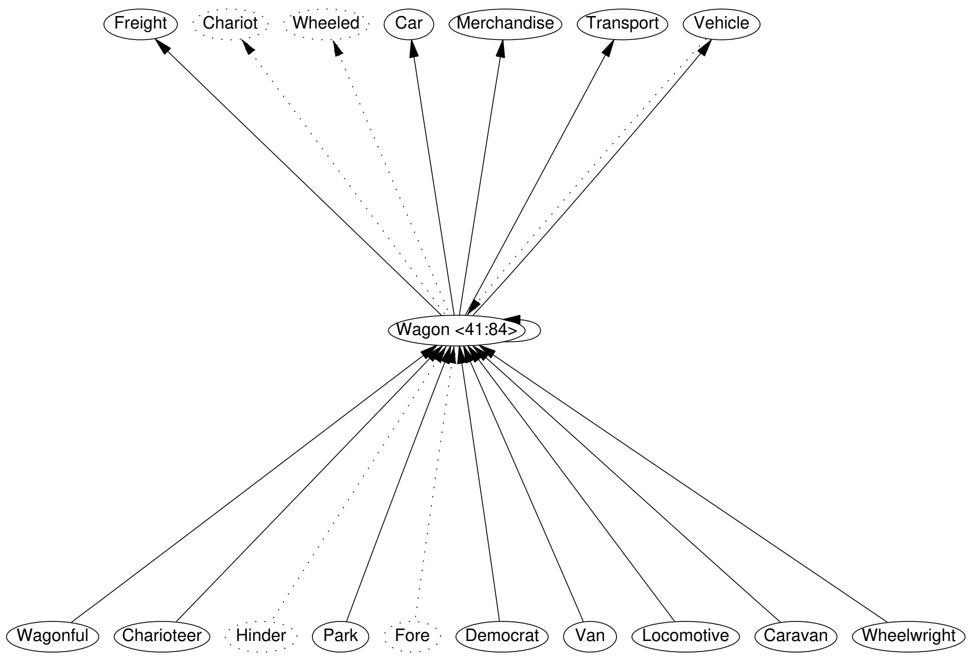


Figure 4.8: Wagon as a Means of Transport

Figure 4.8. Wagon is not a specialization of transport, although transport does subsume it: a wagon is one of a number of forms of transport. We see that terms such as Car and Vehicle, also shown in Figure 4.8, represent a generalization relationship for Wagon. Also, terms such as Charioteer, Caravan and Wheelwright relate to wagon without being specializations. Locomotive is, however, a specialization, and we continue to browse with the chart in Figure 4.9.

The chart for Locomotive illustrates a spectrum of relationships between terms, some of which are altogether unexpected, such as locomotive's relationship to the term Appendix. A glance at the definition of locomotive in Figure 4.10 reveals that a reference to an illustration in the appendix of the dictionary appears inappropriately within the definition field of the term. This unfortunate noise in the source data appears throughout the repository, but is typically outweighed by the amount of useful relationships that emerge from the repository structure. In this case, for instance, the other associated terms all respect some subsuming or entailment relationship to locomotive.

A more careful examination of the definition in Figure 4.10 shows that ArcRank is able to eliminate all stopwords from achieving high ranking, as well as words such as 'especially,' 'one,' 'communicate,' 'motion,' 'bear,' 'convey,' 'draw,' which are either not directly related

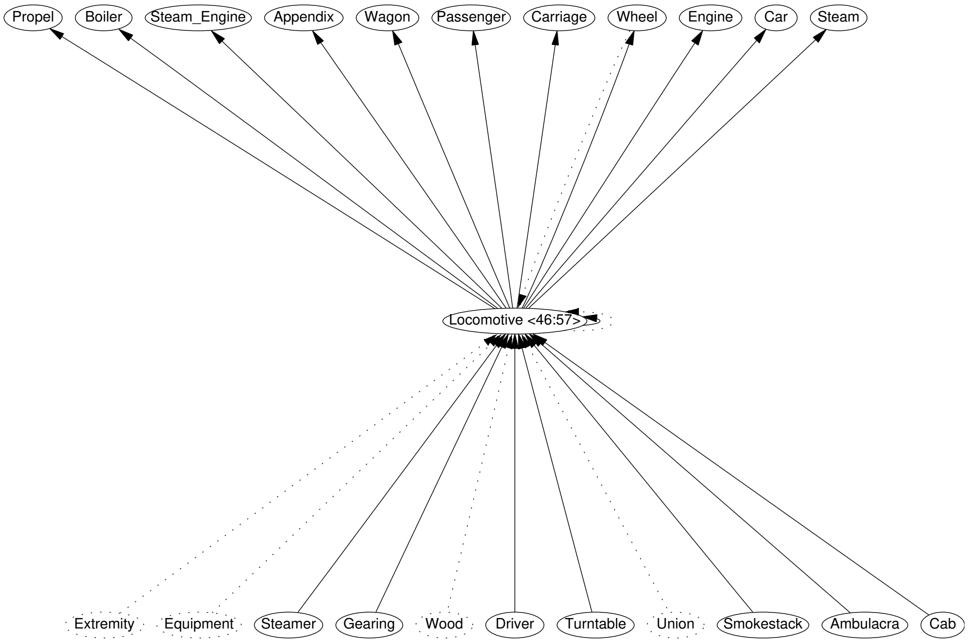


Figure 4.9: Locomotive Specializes Wagon

to Locomotive, or too general. Indeed, we've seen that Bear and Convey appeared at a higher level of abstraction in Figure 4.7.

4.2.4 Charts of Representative Terms

In the examples above we have seen verbs and nouns, and their organization as facilitated by the ArcRank algorithm. In this section we see that the ranking ability of the algorithm extends to adjectives, adverbs, pronouns, prepositions, and beyond these categories, to stopwords. Recall that a unique aspect of ArcRank is that it excludes no terms in its ranking mechanism, so it is able to provide information about stopwords. No other ranking methods handle the stopwords gracefully. We also observe the algorithm handles with no degradation the addition of nodes that represent proper nouns, such as country names.

Figure 4.11 provides a strong example of the clustering of concepts that ArcRank achieves. Light, as well as a multitude of terms relating to darkness, figure prominently in this chart. Note that although adjectives are predominant, nouns and verbs are also present among the important relationships. Such relationships between parts of speech are typical of those that are not captured in WordNet and MindNet.

Figure 4.12 shows how time oriented adverbs cluster around the term Ever. Note again

```
<hw>Lo"co*mo'tive</hw> <pr>(?)</pr>, <pos>n.</pos>
<def>A locomotive engine; a self-propelling wheel carriage,
especially one which bears a steam boiler and one or more
steam engines which communicate motion to the wheels and thus
propel the carriage, -- used to convey goods or passengers,
or to draw wagons, railroad cars, etc.
See <xex>Illustration</xex> in Appendix.</def></p>
...
```

Figure 4.10: Partial Definition of Locomotive

that, in the previous example, the antonym for the term, Never, is also present. The largest category of terms in the chart are adverbs, and two thirds of the terms are time oriented.

Figure 4.13 shows the common adverb Too, and presents us its two common senses, 'also' (as in me too) and 'excess' (as in too much). A class of terms prefixed with *Over-* emerges clearly. This category contains verbs, adjectives, nouns and adverbs, and could not appear in WordNet, since WordNet separates the parts of speech. The problem of disambiguating the senses of terms is left for the next chapter.

Figure 4.14 shows a borderline stopword, since the term It appears in just about 5% of the dictionary terms. Note how well the ranking algorithm performs. The term Pronoun is selected as having a top ranking arc out of 4866 possible candidates. Stopwords as well as the class of terms that emphasize the pronominal aspect of It.

The stopword To in Figure 4.15 still manages to achieve a most interesting top ranking of nodes. The incoming arcs are all from common verbs, and from another stopword, The. The notion of Infinitive and Arrival are the top two outgoing arcs for the term. ArcRank produces useful results where other algorithms are unable to perform, and in categories that WordNet does not cover.

Perhaps the most interesting of all of the categories we examine in this section are the set of special nodes created to represent one class of words that are not defined in the dictionary. Proper nouns, for the most part, are not defined in the Webster's dictionary. However, many of these nouns have a related adjective which *is* defined. For example, Scotland $\rightarrow$ Scottish. In these cases, we add a node to our original definition chart, labeled by the proper noun, and having as a single word definition the related adjective. We use ArcRank to order the terms that use the proper noun to glean insight into the meaning of

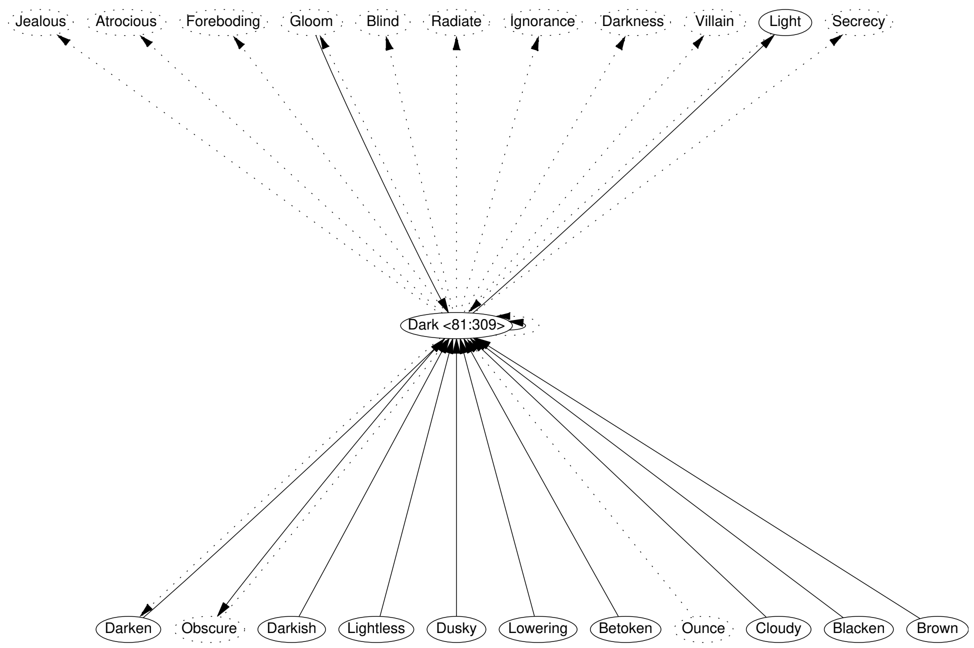


Figure 4.11: Adjective Dark

the proper noun. As we see in the case of Scotland, the terms listed provide information about Scotland. Even Logarithm is relevant, as John Napier, who developed the natural logarithms, was from Scotland.

The ability to add terms to the structure suggests a number of possible extensions to the repository to handle misspellings in the definition terms. For example, when confronted with a word that does not appear as a dictionary head word, such as 'bamana,' we currently ignore the term. Instead, we may take the lexically most similar word 'banana,' using a cost model developed for spelling errors, to be the single word definition for 'bamana,' and just add 'bamana' to the repository.

We have seen how ArcRank is a powerful algorithm for determining the most important relationships between nodes of a directed graph. In the context of the dictionary nexus ArcRank allows us to recognize for each term which are the most relevant terms in its definition, and which are the most relevant terms that us it in their definition. In SKEIN we can therefore exploit the ArcRank values to compute coherency of terms in an information source with respect to the target application. In the next section we suggest applications that build on the ArcRank algorithm. In particular we are interested in using ArcRank to guide the extraction of hierarchical relationships between terms.

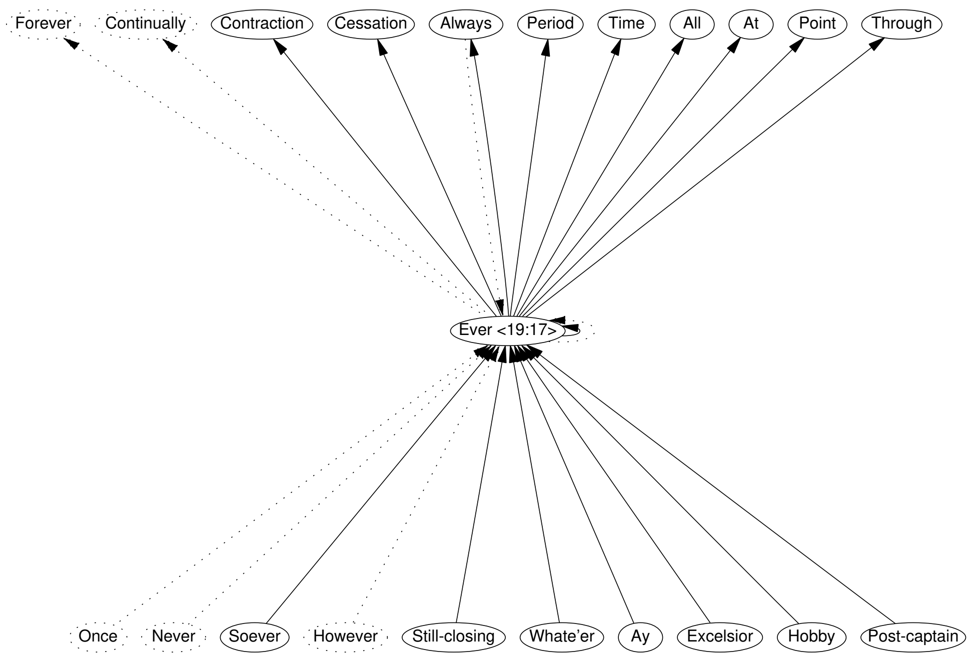


Figure 4.12: Adverb Ever

4.3 ArcRank Applications

Having traveled through a very small sample of the structure of the repository, it becomes clear that the ordering provided by ArcRank is in itself not sufficient to automatically extract the significant terms relating to a given term. An algorithm to achieve this extraction is the basis for the application we are building on top of the repository and is discussed later. As it turns out, the rankings provided by PageRank and ArcRank enable an efficient extraction procedure to make explicit the hierarchical structures embedded in the relationships between terms.

4.3.1 Finding Node Clusters

The ArcRank algorithm identifies the relative importance of nodes to each other. The applications that suggest themselves immediately, therefore, are finding similar nodes, finding nodes that subsume classes of similar nodes, and finding classes of related nodes that specialize a node. Much ongoing research [DH99] is investigating the idea that we may find that nodes are related when they share similar sets of arcs. This research assumes that all arcs are equally important, and therefore is limited in its ability to recognize on which arcs to

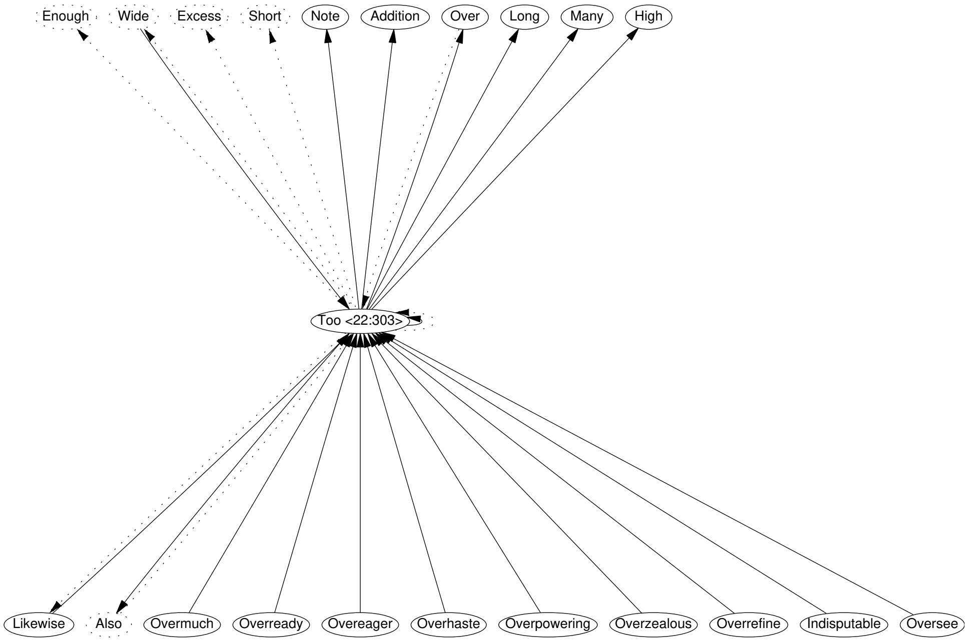


Figure 4.13: Adverb Too

focus. ArcRank provides precisely the information we need to make the distinction between important arcs and less important arcs in order to improve the quality of the extraction of related nodes.

4.3.2 Multi-Source Articulation Support

In Chapter 5 we will develop an example which shows how the repository is used to generate an initial estimate of similarity between terms in distinct information sources. We use the ArcRank values to choose the most likely candidates for similarity between terms in the two sources. The basic approach for this method is to consider the terms from each source as nodes in a bipartite graph. We immediately remove from the graph the nodes which are equivalent on both sides of the graph. The remaining nodes from each side rank the others based on similarity computed according to the algorithm in Section 4.4. We then can use a variant of the stable marriage algorithm to select the best matches between nodes from both sources.

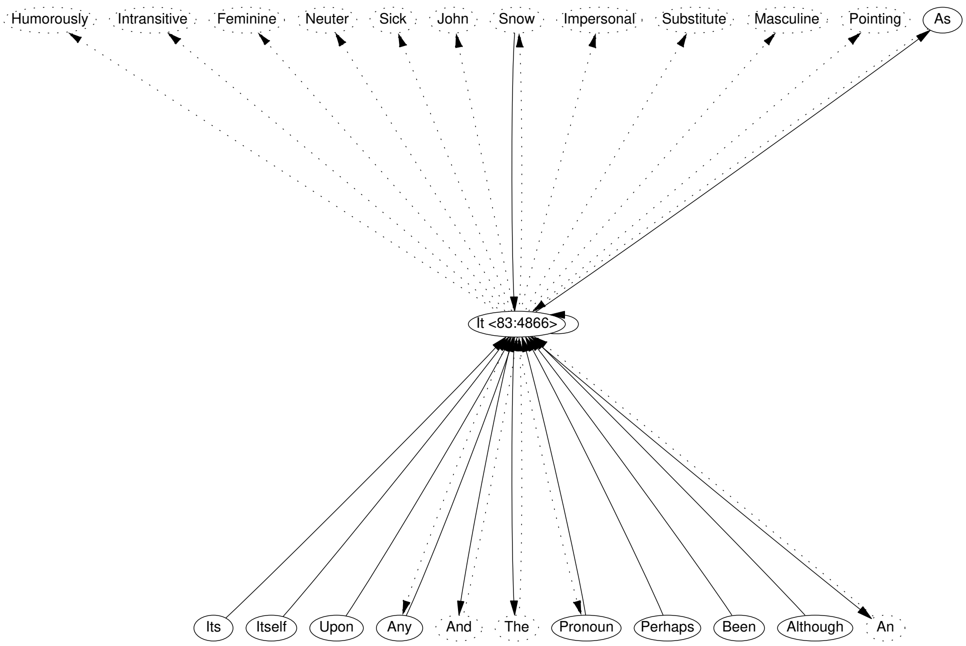


Figure 4.14: Pronoun It

4.4 Term Similarity from Arc Importance

In this section we present the basis for a relationship of kinship or *coherence* between dictionary terms. We use this notion to define a kinship extraction algorithm and the all pairs similarity algorithm. We use these in Chapter 5 to support the generation of new articulations between heterogeneous sources.

4.4.1 Dictionary Definition Pattern

So far, in this chapter we have capitalized on a simple relation between dictionary terms to compute a second, ArcRank, which we posit has a relevancy semantics. Given terms x and y, this amounts to the following:

- y is present in the definition of x
- y is *relevant* to the definition of x

In the next section we use this relevancy assumption to compute kinship between dictionary terms. Our guiding principle is that terms that are coherent to each other must share relevant terms in their definitions and must be relevant to some of the same terms.

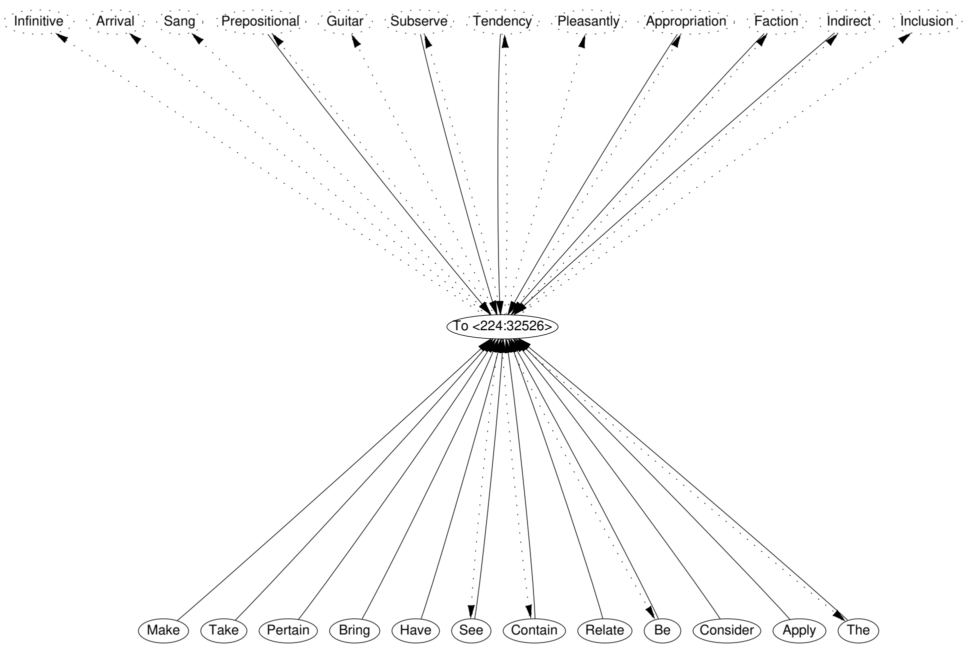


Figure 4.15: Stopword To

4.4.2 Kinship Relationship Extraction

The kinship relationship extraction algorithm is the basic technique described in this section. This is a new algorithm based on the principles of the Pattern/Relation extraction algorithm DIPRE [Bri98], which does not consider hierarchical relationships. Kinship extraction is novel in that it is inherently self-limiting because it applies a thresholding based on ArcRank values. The thresholding reduces the possible number of new candidate nodes at any iteration of the algorithm, to only the most qualified. Also, DIPRE only considers lexical patterns in the the formulation of the problem, whereas our results show that it is possible to use the most general structural relationships between terms as input patterns of the algorithm.

Having a repository with rank relationships between terms, it becomes possible to extract groups of related terms based on the strengths of their relationships. In particular, we are interested in extracting three relationships: *subsuming*, *specializing* and *kinship*. The kinship relationship is a similarity relationship broader than synonymy. We are able to achieve this extraction using a new iterative algorithm, based on the Pattern/Relation extraction algorithm [Bri98], as follows in Table 4.3.

The output of the algorithm computes a set of terms that are related by the strength

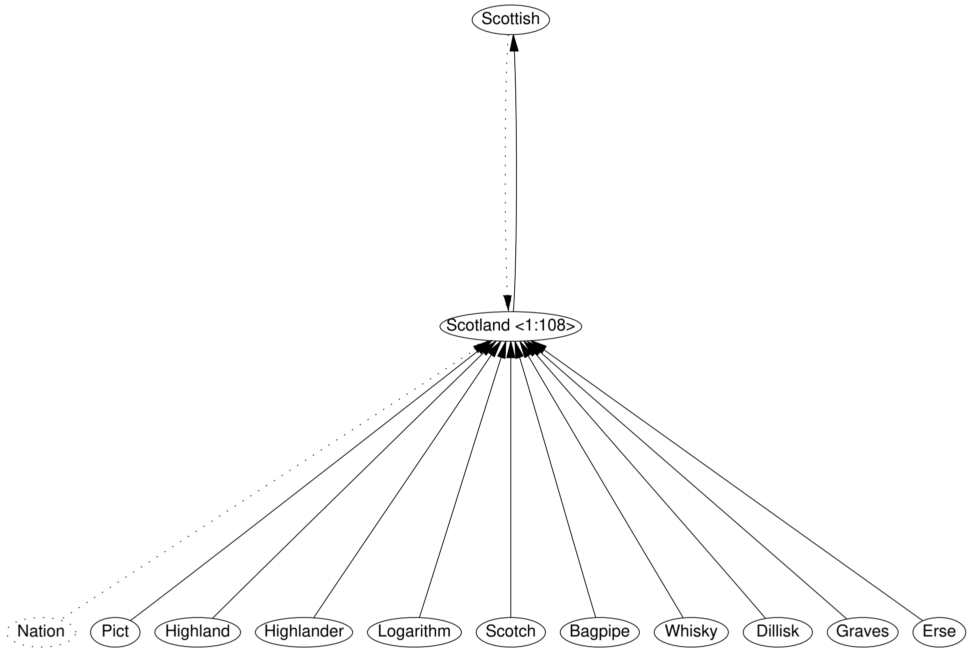


Figure 4.16: Artificial Node Scotland

of the associations in the arcs that they contain. These associations correspond to local hierarchies of subsuming and specializing relationships, and the set of terms are related by a kinship relationship. The algorithm is naturally self-limiting via the thresholds.

This approach allows us to distinguish senses of terms when they engender different structures according to the extract kinship algorithm. Indeed, the senses of a word such as dark in Figure 4.11 may be distinguished by the choice of association with lightless versus foreboding. Also, ranking the different senses of a term by the strength of its associations with other terms allows us to uncover the principal senses of a term.

Table 4.3: Extract Kinship

input graph with ArcRank computed, & seed arc set, *output* local hierarchy based on seed arc set

1. Compute set of nodes that contain arcs comparable to seed arc set
2. Threshold them according to ArcRank value
3. Extend seed arc set, when nodes contain further commonality
4. If node set increased in size repeat from 1.

4.4.3 All Pairs Similarity

This section presents a new algorithm for iteratively computing the similarity of objects based on the importance ranking relationship developed in the previous sections. We show the use of similarity data to support the development of new articulations for interoperation.

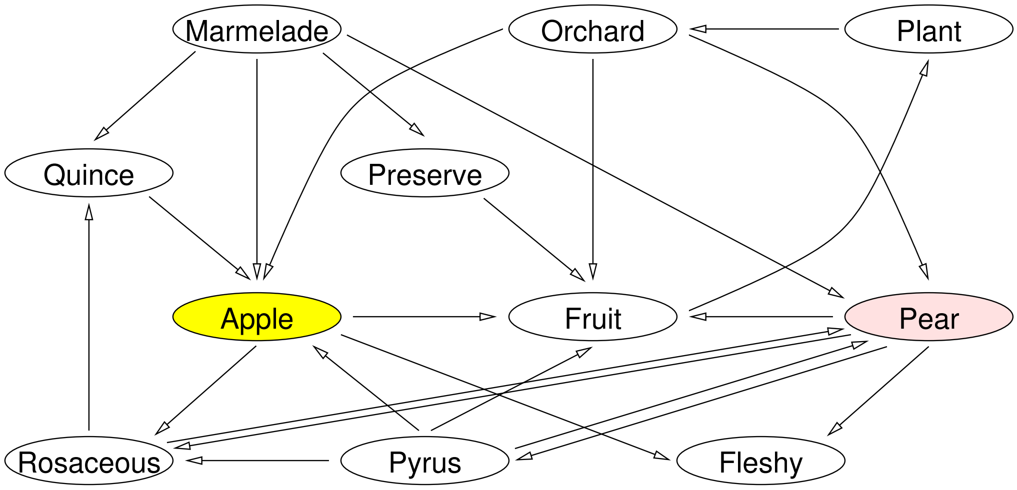


Figure 4.17: Similarity between Apple and Pear

We revisit in Figure 4.17 the fruit subgraph seen earlier with the view of comparing the similarity between Apple and Pear. By inspection, we see that the two terms share, in this small subgraph alone, incoming arcs from Marmalade and Orchard, as well as outgoing arcs to Fruit, Rosaceous and Fleshy. More interestingly, since Rosaceous and Pyrus have similar arcs they have a computed similarity value. Then the paths Apple $\rightarrow$ Rosaceous $\rightarrow$ Pear and Pear $\rightarrow$ Pyrus $\rightarrow$ Apple have a similarity value associated with them, therefore contributing to the similarity of Apple and Pear. In general, every two paths of equal length, with equal end-points, where for every i, the ith items in the paths have a non-null similarity value, contribute to the similarity value of the endpoints of the paths.

In order to do the actual computation of the similarity between Apple and Pear we define a few new terms. The *Usage arc importance Vector* for a term x is a vector of all arc importance values of terms that refer to x. For example, UV_{pear} is the vector of arc importance values for the terms Marmalade, Orchard, Pyrus and Rosaceous which refer to Pear. The usage arc importance vector of a term is easily determined from its backlinks. Similarly, the *Definitional arc importance Vector* for x is the vector of all arc importance values of terms referred to by x. For example, DV_{apple} is the vector of arc importance values

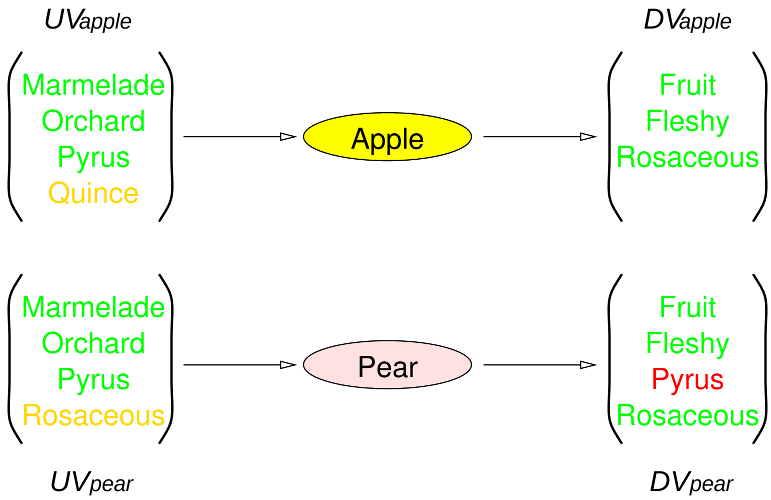


Figure 4.18: Similarity Computation for Apple and Pear

for the terms to which Apple refers. Figure 4.18 depicts the vectors for both Apple and Pear. Given these vectors we now have a tool for computing the similarity of terms. The basic formula for computing the similarity between Apple and Pear is given by the equation below:

$$Sim_{apple,pear} = (UV_{apple} \bullet UV_{pear} + DV_{apple} \bullet DV_{pear})/2 \tag{4.1}$$

In the equation multiplication represents the dot product of the two vectors. The algorithm which we develop to use this formula has one additional feature, based on the following observation. The similarity relationship is partially transitive in the following way: When u and v are similar and u are adjacent to x and v are adjacent to y, then x and y should also be similar. Referring back to Figure 4.17, we see that Quince and Rosaceous should be similar because Quince is adjacent to Apple, Rosaceous is adjacent to Pear, and Apple and Pear have a similarity score. This notion of similarity is also reflected in the lightly shaded entries in UV_{apple} and UV_{pear}. The problem remains of how to recognize and pair up these partially similar terms in the vectors. The solution is to perform substitutions in the vectors based on the stable marriage algorithm [Knu97]. A stable marriage algorithm pairs items up from two sets, based on stated preferences, in such a way that no swapping of item pairs would improve the pairing's preference score. For all of the terms that don't match exactly in the UV vectors, compute the best matching ones based on their current similarity values. The stable marriage algorithm guarantees that no swapping of substitutions of term pairs will produce a better overall matching. Once we run the stable

Table 4.4: All Pairs Similarity

input: directed graph G, *output*: ranked node lists s

For each graph node j
  Make adjacency list representation of input a_j
  Compute reverse adjacency list representation of input (backlinks b_j)
  x_j = union of all adjacency lists containing the node $-j$
  For each node k in x_j
    $s_{0_{j,k}} = 0$
  $s_{0_{j,j}} = 1$
`simchange` $= |s|$
While `simchange` $>$ `threshold` (round i)
  For each graph node j
    For each node k in x_j
      $m_{out_{j,k}}$ =stable marriage$(a_j - a_k, a_k - a_j, s_i)$
      $n_{out_{j,k}} = |a_j \cap a_k|$
      $v_{out_{j,k}} = m_{out_{j,k}} + n_{out_{j,k}}$
      $m_{in_{j,k}}$ =stable marriage$(b_j - b_k, b_k - b_j, s_i)$
      $n_{in_{j,k}} = |b_j \cap b_k|$
      $v_{in_{j,k}} = m_{in_{j,k}} + n_{in_{j,k}}$
      $l_{j,k} = |a_j| + |a_k| + |b_j| + |b_k|$
      $s_{i_{j,k}} = (v_{out_{j,k}} + v_{in_{j,k}})/l_{j,k}$
  `simchange` $= \sum s_{i_{j,k}} -$ `simchange`

marriage algorithm, and perform the suggested substitutions, we can compute an updated similarity value for any pair of terms. This algorithm contains the same notion of iteration as the PageRank algorithm does.

Table 4.4 is a pseudocode description of the all pairs similarity algorithm. The stable marriage algorithm uses s as the preference ranking between elements of the two input sets. A similarity value of 1 between nodes implies perfect similarity, while the value 0 implies no relationship at all. The initial state of the algorithm is that every node is perfectly similar to itself, and itself only. The algorithm is monotonic and converges within $\log n$ iterations, for a total cost of $n \log n \log^2 k \log\log k$, where k is the average number of nodes in an adjacency list. In the OED nexus $k = 16$ while in the Webster's nexus $k = 15$, giving a runtime on the order of $32n \log n$. The algorithm also guarantees that nodes that have no adjacent nodes in common have similarity 0, and nodes whose adjacency lists are identical have similarity 1. Another desirable property is that perfect similarity is transitive. If two nodes have adjacency lists whose nodes are perfectly similar, then the two nodes themselves are also perfectly similar.

All pairs similarity gives us the means to determine similarity in diverse information sources. It does not rely on lexical similarity, but rather on similarity of context as provided by the dictionary definitions extracted into the nexus, and prioritized by ArcRank. We use the results of all pairs similarity in the SKEIN system to perform similarity computations.

4.5 Algorithm Review

In this chapter we have presented the ranking algorithms which form the basis of all of our computations over the dictionary repository structure. We compute the flow over each arc in the repository, as it approaches a steady state, and compute the arcs' relative importance to be the proportion of the destination's node's PageRank that is contributed by the arc. The arc importance values are ranked for each node, and the rankings are used to compute an ArcRank value for each arc of the repository. ArcRank is the basis for the all pairs similarity algorithm between nodes of the repository, and also for the initial similarity evaluation of nodes from differing sources for which an articulation is being constructed. The all pairs similarity algorithm can be computed in $m \log n$ time where m is the number of arcs and n the number of nodes. The other algorithms are linear in the size of the number of arcs of their inputs.

Chapter 5

SKEIN System Infrastructure

Chapter Outline

In this chapter we provide a more detailed description of the algebraic infrastructure for the dissertation. Recall that in Chapter 2 we presented only a brief description of the object model AMO and its associated rule language AMORL. It is the case, however, that the thesis work would not be possible without the underpinnings described in this chapter. We did not have the opportunity to fully flesh out the algebra as proposed for SKEIN. We were able to fulfill the third hypothesis of this dissertation which concerns interoperation cost reductions from using SKEIN. We achieve this result here to the extent that we used the algebraic operators to develop the nexus and that we based them on the results of the ArcRank and similarity algorithms. The algebraic operators described here, three of which were extensively used for the thesis work, are presented in the sections that follow.

We begin with a discussion of semantic context, followed by a listing of algebraic operators considered in the context of the thesis work. We continue with an in-depth presentation of these operators.

5.0.1 Semantic Context

We motivate the need for context by observing that there is no global notion of consistency of information. Models of knowledge that are appropriate for one application may be useless for another. Referring back to the introduction, the database that links my name to diving, is perfectly useless as an advertising tool. However, it is perfectly adequate for myself, my coach, and my teammates, because we know to qualify diving as *springboard diving*.

Identical terms in separate sources will invariably have differing semantics, while distinct terms, even within the same source, may have equivalent semantics. What we desire is the ability to specify that the semantics of the objects relevant to an application are locally consistent and free of mismatch.

We define, following [Guh91], contexts to be objects that encapsulate other objects. Contexts assert the validity of statements about the objects they encapsulate. In other words, given an appropriate set of statements about its objects, a context provides guarantees about their consistency. Since we use contexts to model knowledge obtained from diverse sources for application-specific uses, we are concerned with two specific relationships: *coherence* and *similarity*. The former expresses the relevance of source information to the target application, the latter identifies equivalent and mergeable objects between different sources. While the two relationships resemble each other, distinguishing the two is important for maintenance and scalability. This distinction is motivated, for example, in our earlier work on Ontology composition [JPVW98]. Because we assume sources are autonomous, they may change at any time. In particular, as the number of sources grows, the likelihood of change at any time increases dramatically. By distinguishing coherence and similarity we are able to separate changes of a source that affect their relevancy to our application from those changes that affect their similarity to other sources with which we combine them.

In the SKEIN system, contexts are an object whose value consists of a ruleset and a sequence of objects represented by the ruleset. As implied by the previous statement, object values are a sequence of values, both of primitive and object types. The ruleset itself is an object whose interpretation defines other objects. The ruleset transforms source knowledge into an object set that meets the consistency requirements of the target application. The consistency guarantee, as embodied by a coherence expression, is written in the AMORL rule language defined in Chapter 2. In the next section we present the operators of the ontology algebra in more detail.

5.1 Operators

In this section we review the operators that comprise the algebra. We also detail the application that motivates the inclusion of each of the operators in the algebra. We begin with unary operators, which all transform a source according to a coherence measure. The binary operators, in contrast, take a similarity measure to combine information from two sources.

5.1.1 Unary Operators

We consider four unary operators. The two operators `S` (`Summarize`) and `G` (`Glossarize`) maintain source information, and wrap it in a new object or set. The two others, `F` (`Filter`) and `E` (`Extract`), reduce the source according to an additional predicate. To preserve composability of the algebra, these operators take a directed graph as input and return one as output.

Summarize

We've presented `Summarize` or `S` in detail in Sections 3.2.3 and 3.3.1. `S` is the canonical unary operator of the algebra. `S` creates an object that encapsulates the information of the source, and populates the object with results of an aggregation operation over the source information. The application that motivates the existence of the `S` operator is data classification. The aggregation over the source data effectively groups the source into equivalence classes.

Glossarize

The `Glossarize` or `G` operator, which we detailed in Section 3.2.1, creates an object that contains the set of objects from the source, without any of the object substructure. `G` effectively flattens the source data into members of a single set. `G` is motivated by the need to list all terms that are subordinate to an object.

Filter

The `Filter` or `F` operator reduces the instance objects from the source data according to a selection predicate. At its most restrictive, this operator returns only the schema structure of the source. It is the complement to the `G` operator. `F` is important in reducing the size of a source for verification and validation purposes.

Extract

As we defined it in Section 3.2.1, the `Extract` or E operator reduces the schema objects, as well as corresponding instances, from the source data according to a selection predicate. Wrappers that build on existing wrappers use E to present a reduced view of the original wrapper.

5.1.2 Binary Operators

As with the unary operators there are four binary operators. M (`Match`) and B (`Blend`) maintain the source data as they combine them. The two operators I (`Intersect`) and D (`Difference`) reduce the sources. These four operators take two input directed graphs, and return one as output.

Match

The prototypical binary operator, defined in Section 3.2.1, `Match` or M returns the information from both sources, along with a new object that contains a sequence of pairs of references to matching objects in both sources. Where matching objects differ in name or granularity, new objects are created as necessary to mark the transformation.

Blend

The `Blend` or B operator extends matching objects from both sources. The transformation adds the attributes to each object in a source that are present in the other. The effect of this operation is to map a copy of all the objects reachable from the matching objects in one source onto the other source. The need to extend the schema of objects with the attributes they contain in other sources motivates the addition of B to the algebra. The resulting objects represent blended subclasses of the original source objects.

Intersect

The `Intersect` operator or I returns only the portions of the sources that match. Both copies of the matching objects are returned, since the objects that mark heterogeneity of name, granularity or reference differ between the two sources. The common application for I in practice is the identification of common schema between sources. I returns objects that conform to the schema defined in both sources.

Difference

The `Difference` operator or D returns only the portions of each source which are unique to it. In effect, this corresponds to a symmetric difference between the two sources. The substructure supporting the differing objects in each source (those objects and attributes that keep the graph of the source information connected) is also preserved. In practice, D is used to determine the semantic distance between sources. The cost of transforming the differing objects is in fact the measure of this distance.

5.1.3 Operator Semantics

The eight operators above take semi-structured object graphs as parameters and return them as results. The unary operators are augmented by a coherence expression that defines constraints on the objects transformed by the operator. Binary operators use a similarity expression to specify constraints on the objects that match between sources.

5.2 Semantic Consistency

Before defining the relationships below, it is necessary to define the unit of semantic consistency we use throughout. As we have seen above, there is no meaningful notion of global semantic consistency. Recall that the object model we defined above is structurally related to HTML. In a graph as large and heterogeneous as the World Wide Web, we must use heuristics to define semantically consistent subgraphs. For example, content based heuristics such as: pages located at the same server, having the same URL prefix, pages using relative addressing, or sharing the same style sheet, are candidates for being considered semantically consistent. With XML, document type definitions (DTDs), determine a set of consistent documents, as long as the DTDs are properly used. The types of heterogeneity within documents that we need to address are as follows:

- vocabulary differences

  term differences head of state is president vs. chancellor

  term definitions president is ceremonial vs. powerful

  term polymorphism officials are simultaneously ministers & parliament members

- structural differences

term multiplicity government is Commons & Lords vs. Parliament

term relation ministers are in parliament vs. cabinet

term repetition council of state is separately government & prime minister's office

In practice, even single page documents contain inconsistencies and incorrect data. Therefore, we do not define an a priori measure of semantic consistency at any granularity. Instead, we define a coherence measure, which expresses a data source's relevance to an intended application. Only data sources which enforce the same coherence measure are considered semantically consistent with respect to an application.

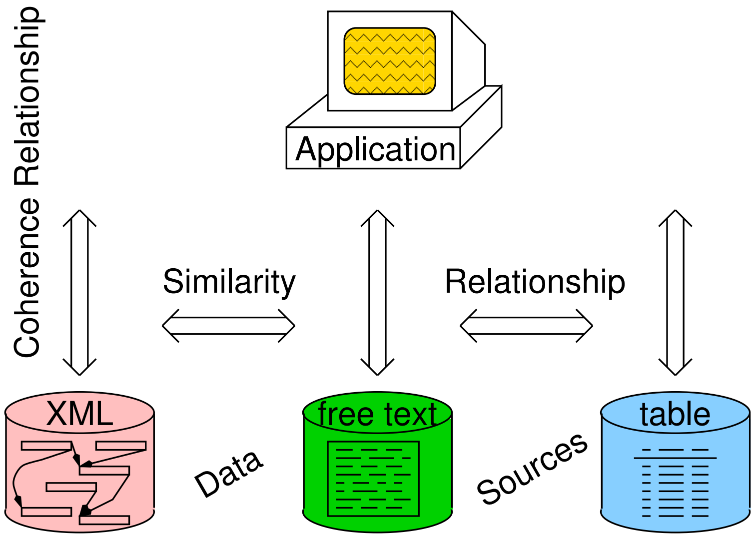


Figure 5.1: Relationships between Sources and Target Application

5.2.1 Coherence Measure

The coherence measure is a binary relationship between a data source and a target application of the data. It states explicitly the portion of the source that is relevant to the target application. Data which is only partially useful due to errors or incompleteness is also expressed in the measure. In Figure 5.1 the coherence measure is represented by the arrow in the vertical dimension. We start with a target vocabulary, small or large, which is a set of terms we know are relevant to the target application. Terms in the source data that have high ArcRank scores relative to the target set are coherent to the target application.

For example, referring back to the graph of Figure 4.17, Orchard is coherent to Apple, Fruit and Pear.

In SKEIN we write the coherence measure as a script of the primitive operators defined in Section 2.1.4 above. In the two extreme cases The empty script represents no relationship between a source and the target application. A script creating a single object that encapsulates the entire source is equivalent to accepting the entire source contents for the target. A script that establishes a coherence measure is also called *coherence expression*. In Section 5.3 we give a detailed example of an iterative use of the algebra to establish a coherence expression between an on-line Webster's dictionary and an application that graphs its structure.

When multiple information sources require different coherence measures to be brought together within a single application, we do not assume that they are mutually consistent. Any blending of information from differing sources requires additionally the application of a similarity measure.

5.2.2 Similarity Measure

The similarity measure is a binary relationship between two data sources. It identifies in two sources, the objects which can be considered equivalent, as well as those objects which may be merged for the purposes of the target application without being identical. The horizontal arrow in Figure 5.1 corresponds to the direction of the similarity relationship in our system. We define similar terms as being terms that share coherent terms. For example, since both Pear and Apple in the OED are coherent with Fruit, then Pear and Apple are similar.

The similarity measure is also represented by a script, or *similarity expression* of the primitive operators in our system. The measure is the empty script when there are no matches between sources. Otherwise, scripts for a similarity measure consist of an object containing a set of references for each matching pair of objects in the sources. We show in Section 5.3 the use of the `Match` operator to create and refine a similarity measure between two government websites.

5.3 Wrapper Semantics

The `Summarize` operator is a unary operator that transforms source data based on a predicate which corresponds to a coherence measure. As described in Section 5.2.1 the coherence measure is not an ideal coherence relationship, but rather an approximation of the ideal. We need a coherence relationship to apply the `Summarize` operator, but it is impossible to establish such a relationship prior to applying the operator. This paradox is what compels us to consider approximations of the relationship. Indeed, we use approximations to bootstrap the construction of better approximations. The algorithm establishing a coherence measure must therefore be iterative and begin with an initial coherence measure. This initial coherence measure represents a first order approximation of the coherence relationship. Figure 5.2 shows such a first order approximation for the dog articulation we presented in Section 2.1.2. In this approximation, only the objects that have an exact lexical match are joined in the articulation.

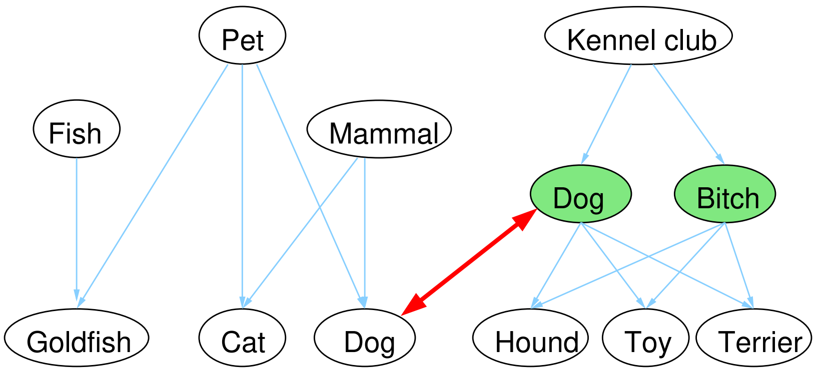


Figure 5.2: Dog Articulation Approximation

At each iteration, an analysis of the outcome of the previous `Summarize` operation allows a refinement of the coherence measure to account for newly recognized exceptions and misclassified data. The iterative refinement continues, until the coherence measure approximates the ideal, within the tolerance level of the application that requires the data. For the dog articulation, we can compute a new approximation by matching all objects that have an arc set identical to the objects in the original match. This new approximation produces the articulation of Figure 2.2 on Page 40. Each application of the operator keeps statistics on the performance of the coherence measure. If an application of the operator produces an inferior result, an exception occurs. If the result of a prior application of `Summarize`, or S for short, is acceptable it will be used. Otherwise, a follow-up round of refinement begins. In this way, changes to sources which affect the performance of S are detected and signaled. Note that the procedure for maintaining the coherence measure is identical to the initial creation and refinement of the measure.

5.3.1 Wrapper Mediation

Systems such as Strudel [FFLS98] provide the ability to restructure consistent pages within a website. Ultimately, we would also like to structure pages with similar content across websites, in order to more easily browse the pages of interest. As an example of sites we would like to browse in a similar fashion, we chose the government websites of NATO members and their partners. Figure 5.3 below represents a partial set of pages and links from the Finnish government's website. The shape of the nodes serves as a visual aid in relating this graph with Figure 5.4. Similarly shaped nodes in the two figures are considered similar by the NATO website, that is they link to them from the same table entries.

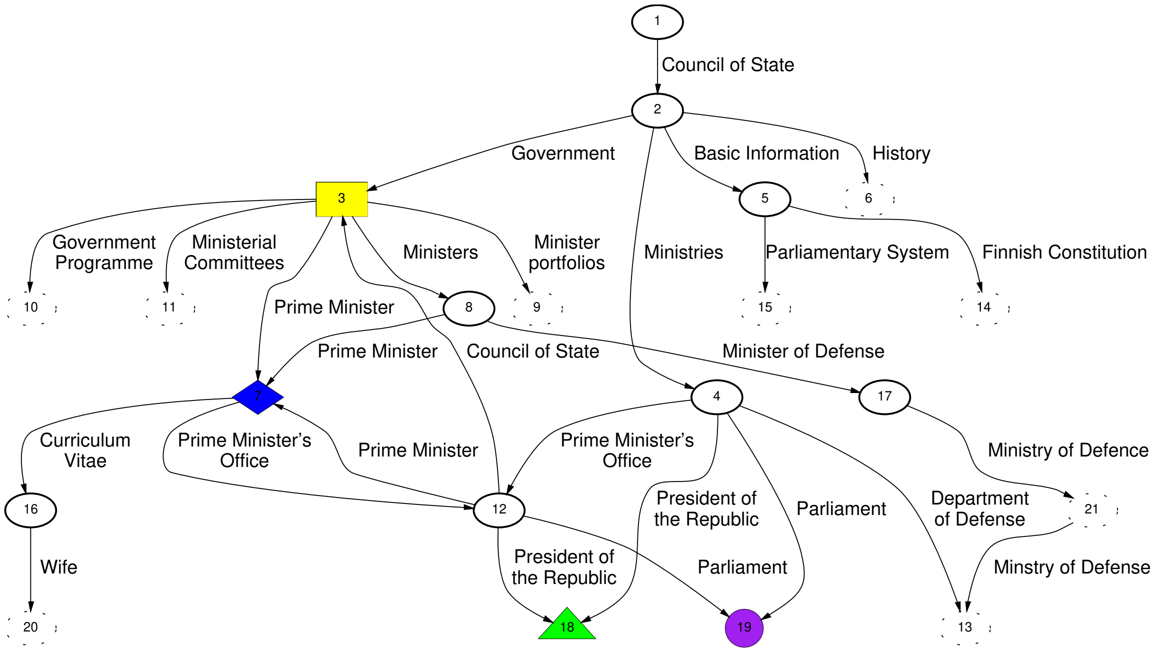


Figure 5.3: Partial Graph of Finnish Government Website

5.3.2 The Match (M) Operator

The Match operator takes two graphs and finds objects that correspond to each other based on a similarity measure that indicates how the correspondence is to be determined. We will highlight the matching using two object graphs obtained from the websites of NATO countries using the S operator defined in Section 5.3. Websites can be thought of as structured as a graph with a root page which has links to other related pages. The labeled graph structure of each website is constructed where each page is a node and all the links found on the page are modeled as outgoing arcs. Each arc is labeled with the text found along with the link, which describes the contents of the pages that the anchor point to. Each node in the graph corresponds to a web page and is assigned a term, based upon the tags on its incoming arcs. The matching of the nodes is based on the similarity of labels, where similarity is determined based upon the similarity measure. Typical mismatches that exist in such object graphs are as follows:

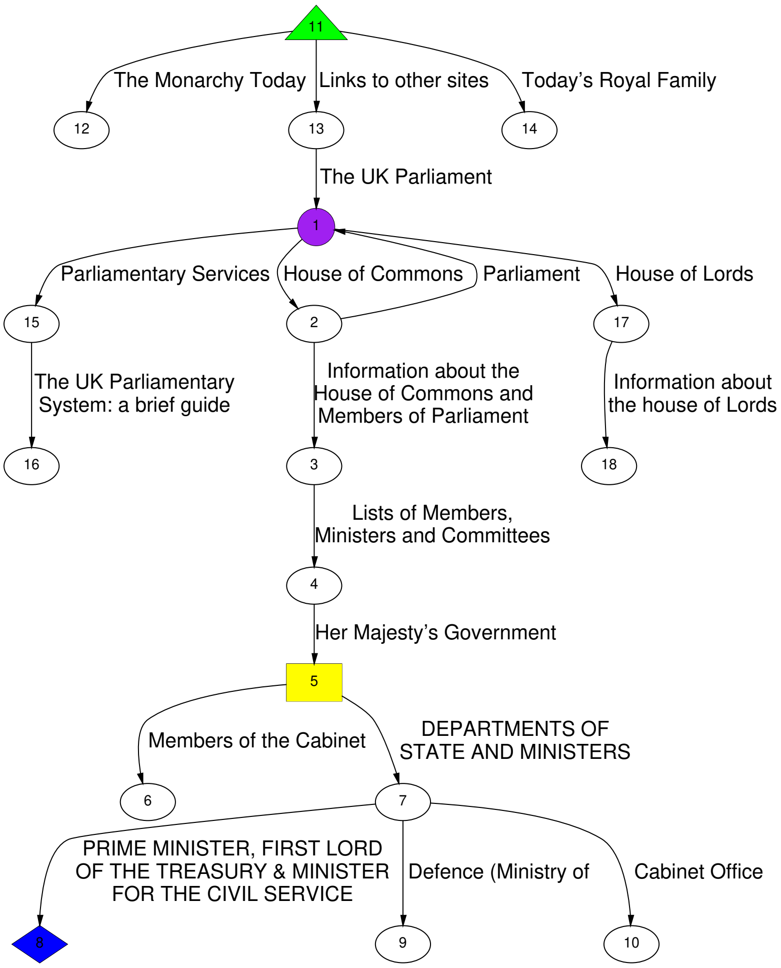


Figure 5.4: Partial Graph of U.K. Government Website

Structural Mismatches : these types of mismatches occur when the same term in one source matches multiple terms in another and causes one node in a graph to match with many in the other. For example, we see that the Prime Minister of the U.K. in

Figure 5.4 is simultaneously Lord of the Treasury. The Finnish Prime Minister is not Treasury Minister, which is the closest counterpart to Lord of the Treasury. The node for the British Prime Minister should thus match two nodes in the Finnish graph.

Instance Mismatches : these mismatches occur because in one source an instance of a class is not an instance of the same class in the second source. The closest case of such mismatch in our figures is that Parliament in Figure 5.4 is a parent to both the House of Commons and the House of Lords, while Parliament refers to a single body in Figure 5.3.

5.3.3 Rule Based Semantic Mismatch Resolution

In the next subsections we describe heuristics which we apply to the establishment of a similarity measure between the government Web page graphs of Finland and the U.K.

Context Identifier Tagging

Our application may require that we differentiate Parliament in the U.K. and Finland governments, and associate the Finnish Parliament with the British House of Commons. We must distinguish the semantics of the same term used in two different graphs. The basic technique allows a preprocessing of terms so as to distinguish them later when performing a lexical comparison of terms. For instance, we may edit the Finnish 'Parliament' to 'Parliamentary Body'. We can also consider a more general-purpose heuristic which indicates "if TermX is a descendant of Government and TermY is not, then do not match TermX to TermY." Such a heuristic generalizes a structural observation from our graphs.

Context Identifier Removal

Matching is performed based on a similarity criterion. In our running example, we want to match the government nodes of our two nation's graphs, e.g., we want to match the node labeled Her Majesty's Government in Figure 5.4 with that labeled Government in Figure 5.3. Therefore, a set of edit operations can be supplied that strips the labels such as prefixes such as Her Majesty's and thereby enable the matching.

Term Mismatch Resolution

These operations simply express that two terms are semantically related and should match. Examples from our two figures include rules such as: `(Match Monarchy President)` and `(Match Defence Defense)`. The first one indicates that we intend to match the head of states though they might be named differently.

These can be more complex than such simple lookup rules if we have a theorem prover at our service, e.g., the two labels 'The UK Parliamentary System' and 'Finnish Parliamentary System' can be matched using a more complex rule like:

```
(Instance-Of Country UK)
(Instance-Of Country Finland)
(<= (Match ?Country1 ?Country2)
        (and (Instance-Of Country ?Country1)
                 (Instance-Of Country ?Country2)))
```

Our system should generate the tuple `(Match UK Finland)`. The fact that U.K. and Finland are two countries can either be explicitly specified or can be obtained from standard knowledge libraries. The fact that we are interested in matching countries is indicated by the last statement.

The second type of operations are needed for differences in spelling in British versus American English, for example. The second type of mismatch will in certain cases be resolved in the preprocessing stage wherein we indicate root-words of words, e.g., `(Root-Word Parliamentary Parliament)`. The preprocessor substitutes the word for its root-word before proceeding with the matching. In the absence of any of such case specific rules, we may proceed with algorithms like Porter's stemming algorithm. As with any automatic process, stemming may result in some spurious matches or cause some matches to fail, whereas the rules let one dictate exactly what we want to match and what not to match.

Irrelevant Match Resolution

A preprocessing stage removes stop-words like 'of' and 'the,' which can cause spurious matches, either by looking up a table which the user can supply or by using an IR metric which assigns weights to words based on their occurrences across nodes and in a particular node. Words that are very frequently used across nodes are assigned very low weighting.

The matching process then computes a weight which indicates the degree of match and these low weighting words do not contribute to the match and the process is as good as having deleted them. An alternate source of spurious matches is the use of stemming algorithms. Words such as minister and ministry might have been stemmed to minister and therefore result in a match, despite our wanting to preserve the difference between the two. Sanity checking heuristics, which state explicit mismatches `(Mismatch Minister Ministry)` or a more general `(Instance-of Person X) and (Instance-of Office Y) => (Mismatch X Y)` help us in preserving the semantics we desire.

5.4 SKEIN Performance

The SKEIN system has been used to generate matches between the above two NATO government sites. The first attempts to match up nodes from the two graphs produced the results in Table 5.1. The match was produced by applying the script in Appendix B.6 to the text of the graphs extracted from the web pages.

Table 5.1: NATO matching results

UK graph	Finland graph
The UK Parliament	Members of the Parliament
Information about the House of Commons and Members of Parliament	Statistical data on Parliament
The Government	Government
House of Commons	Council of State
The UK Parliamentary System: a brief guide	Finnish Parliamentary System
House of Lords	Council of State
Desc. of Ministry of Defense	Department of Defense
Prime Minister, First Lord of the Treasury & Minister for the Civil Service	Prime Minister
Defence (Ministry of	Ministry of Defence
Information about the house of Lords	Basic Information
Her Majesty's Government	Government Programme
Cabinet Office	Prime Minister's Office
DEPARTMENTS OF STATE AND MINISTERS	Minister of Defense
Members of the Cabinet	President of the Republic

These results contain only four false positive matches. The final four, separated from the others by a horizontal line, are the false positives. All but one of the correct matches that the human expert produced are present in the results. What's more, the first refinement to the system that incorporated graph structure into the computation produced no false positives at all, but also reduced the total number of correct matches to 70%. These initial results are extremely promising, especially since computation times grow sub-linearly in the product of the size of the input data sets. The SKEIN system, coupled with the word nexus has great promise in cutting down the amount of effort that goes into producing mediators between such heterogeneous information sources.

In this chapter we have seen how the use of our algebraic infrastructure makes it possible for the SKEIN system to reduce the cost of articulation development. Two websites designed by different people in different countries, but containing related information are matched up with 70% accuracy using the SKEIN system. SKEIN's initial matches are generated using the output of the ArcRank and all pairs similarity data that we described in Chapter 4. These algorithms operated on the nexus whose construction we presented in Chapter 3. ArcRank and all pair similarity scaled well as we quadrupled the size of the nexus by operating on the OED rather than the Webster's dictionary. The nexus itself is bootstrapped from its source data using the object model and rule language developed for the SKEIN system and presented in Chapter 2. Each result builds on the previous one as suggested in Figure 1.10 and fulfills one of the hypotheses presented in Section 1.2. The next chapter suggests future directions to take this work, based on the results we have achieved.

Chapter 6

Conclusions and Future Work

In the preceding chapters we presented an efficiently computed structure, the word nexus, which explicitly models the relatedness of terms in the dictionary. We developed algorithms that use the nexus to express the coherency of the dictionary terms based only on their definitions, and the similarity of terms based on how many coherent terms they share. We used the output of these algorithms to reduce the cost of determining similar pages between two European government websites linked from the NATO website. We have come full circle in this process. The object model and rule language form the foundations of SKEIN. These foundations were sufficient to produce a nexus from the OED source. The study of the structure of the nexus led us to algorithms for computing coherence and similarity values between dictionary terms. These algorithms allow us to specify the algebraic operators which complete the SKEIN system. Along the way we were able to verify scalability and extensibility results for our work. In the next sections we review how these results fulfill the hypotheses we defined in Section 1.2, and consider future directions for our research.

6.1 Novel Word Nexus

Chapter 2 introduced the object model AMO and the rule language AMORL that form the two fundamental layers underlying the articulation algebra we defined in Chapter 5. AMO is a simple object model which is general enough to easily represent Web pages, as well as database relations, and plain text. It also is sufficiently flexible to model more complex object models with strong typing and inheritance. AMORL enables the use of objects as proxies for objects and values taken from individual sources as well as the matching of objects from disparate sources. We exploited the simplicity of AMO and AMORL to bootstrap the construction of a word nexus.

In Chapter 3 we presented our case study demonstrating advantages of the algebraic approach to ontology management. An on-line dictionary represents an ideal test bed for the use of our ontology algebra on real world problems. We used AMORL to develop a ruleset that converted the dictionary definition into the nexus structure. We showed how the consistency guarantees established for the ruleset were essentially preserved in the face of substantial changes to the source data.

6.2 Scalable Nexus Algorithms

In Chapter 4 we showed an important application over an information source that relies heavily on the articulation algebra. We showed how we derive a ranking algorithm that assigns an importance to the arcs of a large directed graph, based on a prior ranking of the nodes of the graph. We proceeded to determine subsuming and kinship relationships between nodes, using an algorithm which capitalizes on these ranks. These techniques represent an important step in extracting hierarchical structures from graph structured information in a way that respects the internal structure of the data.

The development of the ArcRank algorithm was the culmination of an effort to visualize the complex and vast structure of the nexus. The early tools we developed to simplify the structure of the graph reduced by almost three quarters the size of the graph we needed to visualize, but the strongly connected cluster or kernel of the graph required an approach that would efficiently and scalably prioritize the arcs in the graph's kernel. ArcRank fulfilled this requirement.

6.3 Efficient SKEIN System

We defined a set of operators for semantic interoperation of heterogeneous information sources and shows how they are applied to develop articulations rapidly. These articulations serve as the mediating glue between existing information and the new applications that need the information. These operators account for the requirements of target applications through a coherence measure and handle semantic heterogeneity of information sources using a similarity measure. These measures approximate the semantic relationships between source and target application on the one hand, and between differing sources on the other. The articulation algebra builds on AMO and AMORL as well as on the nexus to support experts in generating articulations.

The unique feature of the articulation algebra is that it incorporates a form of closure to model realistically the process of mediator creation, refinement and maintenance. By representing this process as a sequence of iterative steps associated with a qualitative measure, it is possible to make specific claims about the value of the information obtained using the algebra. Also, it represents mediator maintenance within the same framework. Any changes of information in the sources, expert qualifications or application requirements are fed back into the algebraic operators as a further iterative step. This iteration may induce further rounds of refinement before an amended mediator is complete.

Our examples show the use of SKEIN on real-world problems, such as the processing of a large source such as the OED. We also demonstrate the matching of terms between NATO government websites. SKEIN's proposed NATO articulation matches are 70% accurate, contain no false matches. What's more, these results were achieved without any fine-tuning whatsoever of the system. We describe the close relationship between the algebra and the process of creating, refining and maintaining wrappers and mediators for our applications. Our experience shows that building wrappers within SKEIN's framework substantially simplifies their creation, as well as improves their maintainability.

6.4 Relevant and Future Work

This thesis work has pioneered a new representation of the process of developing semantic mediation services for the interoperation of disparate information sources. The full development of the SKEIN system remains a work in progress. In its current state, however, it represents a first step in a systematic treatment of semantic heterogeneity within the framework of an algebra over semistructured data. It has opened a number of avenues of further research in the following areas:

1. stopping articulation algorithms at any time without losing results

2. opposing the expert and information sources in the context of game theory

3. mining market baskets with a label in the domain of the basket items

The subsections below consider these areas in turn.

6.4.1 Anytime Algorithms

In Section 1.2 we use a termination criterion to define a minimal quantitative standard for the quality of an articulation. We take a random sample from the instances accessible from the sources through the articulation to determine whether the articulation meets the minimal standard. Therefore, the termination criterion also serves as a confidence measure of the quality of the data.

If we terminate the operator closure at any time, we may collect a few random samples to define a confidence interval, which tells us how reliable our articulation is likely to be over the entire accessible data set. When our applications handle uncertainty in information, we have a new way of using the articulation algebra. Indeed, the operators take on the characteristics of an anytime algorithm, since no matter when we stop the iteration, we are able to associate a confidence interval with the resulting data. Anytime algorithms [HP98] are a class of algorithms that are increasingly being considered for real-time and on-line applications.

6.4.2 Game Theory

The iterative process of mediator development is related to a non-cooperative two player game [Nas51]. The information source represents an adversary to the expert, who must efficiently extract the relevant information from the source. The information source passively resists the expert's efforts. Each iteration in the operator closure represents a turn in the game. The game ends in victory for the expert when the termination criterion is reached. If a fixpoint is reached before the termination criterion, the game ends in victory for the information source. The connection between game theory and the iterative closure employed by the operators of the algebra could lead to a better understanding of the convergence and completion of the operators' execution.

6.4.3 Meta-Data Mining

In traditional market basket analysis, the basket is an aggregation that is separate from the domain of the items in the basket. It could be of interest to label each basket with the name of the most important item in the basket. Of course, we do not have access to the information which tells us what the important item in the basket is, so we must hypothesize what it could be. On the contrary, in the dictionary domain, each definition represents a basket, and the basket is labeled with the term represented by the definition. This characterization gives us two new data mining problems of interest. What new types of data mining algorithms emerge from the labeled basket problem? How does one label an unlabeled market basket problem? There exists some work on mining metadata [WM00] and extracting ER diagrams from flat files [BHA96], but this work appears to focus on small sources only. We have completed some preliminary work on implication rules and clustering rules that investigate the first question. We have an algorithm that extends ArcRank by generating links between the basket domain and the item domain to explore the second question.

6.4.4 Final Thoughts

The process of opening up new ground in a field invariably opens up many more problems to attack than were originally answered. We feel that the above subsections provide the best sign of fruitful results in our work.

Future work aside, this thesis research has produced some results of lasting merit. The OED nexus is interesting for its sheer size compared to any other on-line repository of semantic relationships between dictionary words. What adds to its value significantly is that it has been analyzed at three different levels which corroborate its consistency: first, it clusters terms semantically, as evidenced by such clusters as the one for the term egoism as presented in Chapter 4; second, the macroscopic arc structure of the nexus is preserved for both the Webster's dictionary and the OED, a feature which could not possibly emerge if the nexus were inconsistently constructed; third, the NATO government web page articulation could not statistically achieve a 70% matching success, if the nexus structure was inconsistent. The ArcRank and all pairs similarity algorithms are scalable techniques to compute coherence and similarity scores for nexus entries. The recognition of the coherence problem as an important one in the interoperation of information is another insight gleaned from this work. The OED nexus will continue to live in ongoing research at Stanford, such as RegNet/RegBase [LW01] and OntoAgents [DMW00], and visualization techniques pioneered for the nexus have found commercial implementations at Gigabeat Inc. [Gig00]

Appendix A

Data Format Conversions

Data Conversion Basics

In this chapter we express some of our data formats with XML DTDs to improve the portability of the SKEIN system. We begin with a simple expression of the AMO object model, which forms the basis of SKEIN. The ArcRank definition is the basis for the term nexus, while the Visualization DTD is used in the commercial implementation of our technology at http://www.gigabeat.com/disc/. The nexus definition allows us to export the nexus structure with a minimal amount of tagging overhead.

A.1 AMO DTD

The AMO model is a minimalist object model. The only requirement of an object is that it have an OID, and everything else is an atom. objects may also contain a set of attributes with atomic values and have a value that is a sequence of object references and atoms. This simplicity guarantees that it is easy to instantiate objects from atoms. It is powerful enough, however, to model more complex object systems, albeit inefficiently.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE AMO [
  <!ELEMENT OBJECT (OID,ATTRIBUTES,VALUE)>
  <!ELEMENT ATTRIBUTES (NAME,ATOM)*>
  <!ELEMENT VALUE (OID|ATOM)*>
  <!ELEMENT OID (#PCDATA)>
```

```
  <!ELEMENT NAME (#PCDATA)>
  <!ELEMENT ATOM (#PCDATA)>
]>
```

A.2 ArcRank DTD

The ArcRank type is again, quite minimalist. It in fact represents a special kind of reified arc. If we assume that the source and target nodes described by the DTD are objects, then ArcRank objects are simply reified arcs that have a score associated with them. The point is that ArcRank can be easily computed from AMO objects using the `reify` method of the AMORL described in Chapter 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE ArcRank [
  <!ELEMENT RANK (ITEM)+>
  <!ELEMENT ITEM (NUMBER,NUMBER,RANK)>
  <!ATTLIST ITEM type (source|target) "source">
  <!ELEMENT NUMBER (#PCDATA)>
  <!ELEMENT RANK (#PCDATA)>
]>
```

A.3 Visualization DTD

Although its development was not a part of this dissertation, the patented visualization method used at Gigabeat Inc. and known as an affinity chart was heavily influenced by the SKEIN system. Figure 4.4 on Page 84 illustrates the general features of an affinity chart. The chart's design responds to the need to have a simple method of visualizing and navigating a large graph structure such as the word nexus. The patent underlying the affinity chart covers the visualization of objects based on the strength of their connections to others, i.e., coherence. The coherence can be computed using the ArcRank algorithm. The affinity chart has a compact and easy to understand layout that is clickable, allowing the traversal of the entire data set. A key usability innovation is that it allows the navigation of a directed graph in both the forward and the reverse direction of the arcs. A feature of the layout is that it optimizes the amount of information that can be unambiguously presented in a given rectangular space, where menus would be ineffective. No other visualization systems combine this simplicity and comprehensiveness together with quick graphical updates compatible with any web browser.

The affinity chart contains a central item and one or two lists of items, ordered by strength of relationship to the central item.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE affinityChart [
  <!ELEMENT SIMILAR (ITEM,(LIST)*)>
  <!ELEMENT LIST (ITEM)+>
  <!ATTLIST LIST arm (upper|lower) "upper">
  <!ELEMENT ITEM (SEARCH,NAVIGATION)>
  <!ATTLIST ITEM type (principal|related) "related">
  <!ELEMENT SEARCH (#PCDATA)>
  <!ELEMENT NAVIGATION (#PCDATA)>
]>
```

A.4 Nexus DTD

Finally, we present a sample DTD for a nexus. The primary goal with this DTD is to completely encode the structure using as few superfluous characters as possible. The reason for this is simply the size of the nexus data. Without any explicit formatting at all the nexus is just under 150MB, so it is imperative to minimize the overhead caused by the tags. Below the DTD are a few sample values showing two entries and a directed arc between them.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE Nexus [
  <!ELEMENT E (S)*>      <!-- dictionary entry -->
  <!ATTLIST E
        i ID #REQUIRED   <!-- entry id -->
        l CDATA "">      <!-- entry label -->
  <!ELEMENT S EMPTY>     <!-- dictionary subentry -->
  <!ATTLIST S
        l CDATA "">      <!-- entry label -->
```

```
  <!ELEMENT A EMPTY>     <!-- directed definitional arc -->
  <!ATTLIST A
        s IDREF          <!-- source entry -->
        t IDREF          <!-- target entry -->
        r CDATA "1"      <!-- arc multiplicity -->
        f CDATA "0.0">   <!-- ArcRank score -->
]>

<E i=0 l="dog">
  <S l="hound"/>
</E>
<E i=1 l="mammal"/>
<A s=0 t=1 r="1" f="0.82"/>
```

Appendix B

SKEIN Tools and Nexus Scripts

Script Details

This chapter presents listings of some of the code used to develop SKEIN and the OED word nexus. The first code is a script that is used to segment the original corpus. AWK is preferable for this task because the on-line OED does not contain any carriage returns, and every other text processing tool's performance is highly dependent on handling text line by line. Python is used for the other scripts. It provides the best balance of quick development time, maintainability, and broad functionality provided by its libraries.

B.1 OED Segmentation

This script takes the input and breaks it into chunks containing each at least one head word and one or more definition words. The difficulty is that the tagging of the OED corpus is deficient in many respects. Many definition words are inadvertently left out of the definition tags, and many of the terms have sub parts that are not well delineated.

```
BEGIN {        # Changed 7/19/00 Jan Jannink
  IGNORECASE = 1
  FS = "<DEF>"    # separates definition from other fields
  RS = "</DEF>"    #'</DEF>' is the record separator
  j = ""      #output record
  k = ""      #active record
  m = ""      #output definition content
  n = ""      #active definition
```

```
  q0 = ""      #input record
  q1 = ""      #input miscellanea
  q2 = ""      #input definition
  q3 = ""
  q4 = ""
  q5 = ""
  u = ""
  v = ""
  w = ""
  xx = 0
  yy = 0
  x = 0
  y = 0
  z = 0
}

{            #first define q0, q1, q2
  q0 = gensub("= *", "= ", "g", $0)
  q1 = gensub("= *", "= ", "g", $1)
  q2 = gensub("> *$", "&_", "g", $2)    #prevents bad ALSO trigger
  if (length(k)) {        #k & n initialized at ""
    if (match(q1, "<LF>.*<E>.*<LF>")) {    #check if chunks span Entries
      while (match(q1, "<LF>.*<E>.*<LF>")) {  #break up some chunks
        if (match(q1, "<ET>..*</ET>.*</E>")) {  #if ET tags before /E
          q4 = gensub("</ET>.*", "", "g", q1)  #find more definition text
          q5 = gensub("<ET>.*", "", "g", q4)
          lq = length(q4) + 6      #total length of segment
          q4 = substr(q4, length(q5) + 5)
        } else {
          q4 = ""
          q5 = gensub("<E>.*", "", "g", q1)
          lq = length(q5) + 4      #segment length (length(q4) = 0)
        }
#print "::q4::", q2, "|", q5, "|", q4, "|", z
#print "::err::"
        dochunk(q5 FS q4, q5, q4)    #fix tagging errors in the OED
        q0 = substr(q0, lq)      #avoid duplicate use of text
```

```
        q1 = substr(q1, lq)
#print "::q3::", q3, "|", q1, "|", q0
      }
    } else {
      fixq3(n)          #define q3 from n
      if (length(q3) < 10 && match(q1, "</SE>..*<SE>") == 1) {
        q4 = gensub("<SE>.*", "", "g", q1)  #free text (should be a def)
        lq = length(q4) + 4      #total length of segment
        y = sub("^</SE>", "", q4)
        dochunk(FS q4, "", q4)
        q0 = substr(q0, lq)      #avoid duplicate use of text
        q1 = substr(q1, lq)
      }
    }
    dochunk(q0, q1, q2)
  } else {          #k null, update previous data
#print "::nok::"
    doprint(q0, fixq2(q2))
  }
}

END {  # not guaranteed to be correct, but happens to be so for the OED
  doprint("", "")        #output final values
  doprint("", "")
}

#print "::m::", m
function doprint(q0, q2) {      #global q0,q2 / dochunk q0,q2
  if (length(j)) {        #<=======  OUTPUT result
    print j
  }
  j = k
  m = n
  k = q0 RS
  n = q2
}
```

```
function fixq2(q2) {        #global q2 / dochunk q2
  z = gsub(" *<MPR>.*</MPR>", "", q2)    #clean up q2
  z = sub("<IPR>.*</IPR> *", "", q2)
  z = gsub("<PS>[^>]*</PS>", "", q2)
  z = sub("<SN>[^>]*</SN>", "", q2)
  z = sub("([a-z]*\. <LB*>[^>]*</LB*>)", "", q2)
  z = sub("<LB*>[^>]*</LB*>", "", q2)
  z = sub("<#>[^#]*</#>", "", q2)
  z = sub("<gk>[^>]*</gk>", "", q2)
  z = sub("<QP>.*</QP>", "", q2)    #keep quotations in some cases?
  z = gsub("<[^>]*>", " ", q2)
  z = gsub("   *", " ", q2)
  z = sub("^ ", "", q2)
  z = sub(" $", "", q2)
  z = sub("^[)(_.,;!?':-]*$", "_", q2)    #avoids false sense merging
  return q2
}

function fixq3(n) {        #q3 must always be global var
  q3 = gensub(" *[|] *", " ", "g", n)    #update n => update q3
  z = gsub("[?()']", "", q3)
  z = gsub("   *", " ", q3)
  z = sub("^ ", "", q3)
  z = sub(" $", "", q3)
}

function dochunk(q0, q1, q2) {
  q2 = fixq2(q2)          #clean value of q2
  fixq3(n)          #define q3 from n
  if (index(q1, "<LF>") > 0) {
    z = match(q3, ", ?$")      #value of z IMPORTANT
    if (z == 0) {
      z = match(q3, "Hence$")
      if (z == 0) {
        z = match(q3, "Also$")
        if (z == 0) {
          z = match(q3, "So$")
```

```
        }
      }
    }
    y = sub(" *[_]$", "", q3)
    y = sub(" *[_]$", "", n)      #fix trigger preventer
#print "::z::", q3, z, length(q3) - z
    if (z > 0 && length(q3) - z < 8) {    #value of z IMPORTANT
#print "::so::"
      k = k q0 RS
      n = n "|" q2
    } else {
      z = gsub("[;:.]* *[.,]", ".", q3)
      z = sub(" [sv]*[bi1][.]", "", q3)    #stray 'part of speech' tags
      z = gsub("[,; ][,; ]*", " ", q3)
      z = gsub("   *", " ", q3)
      z = sub("^ ", "", q3)
      z = sub(" $", "", q3)
      z = match(q3, "= *prec\.[^=]*$")
      xx = z > 0 && z > length(q3) - 20    #match to previous word
      z = match(q3, ": see prec\.")
      xx = xx || (z > 0 && z > length(q3) - 24)  #match to previous word
      z = match(q3, "= *next[^=]*$")
      yy = z > 0 && z > length(q3) - 10    #match to following word
      z = match(q3, ": see next\.")
      yy = yy || (z > 0 && z > length(q3) - 24)  #match to following word
#print "::xx::", xx, "::yy::", yy
      if (match(q3, "^etc\.$") || match(q3, "pl\.$") || xx || yy) {
        z = 0
      } else {
        if (length(q3) < 10) {
          z = sub("^[a-zA-Z][a-zA-Z() -]*[a-zA-Z-][\!.;:]$", "", q3)
        } else {
          z = match(q3, "[a-zA-Z]")
        }
      }
#print "::q3::", q3, z
      if (z) {          #normal case
```

```
#print "::norm::"
        doprint(q0, q2)        #<=======  OUTPUT result
      } else {          #exception case
        if (xx || match(q3, "^etc\.$")) {
          x = y + 1
        } else {
          if (yy || match(q3, "pl\.$")) {
            z = sub("= *next[^=]*$", "", n)
            x = y - 1
          } else {
            u = gensub("</LF>.*", "", "g", j)
            v = gensub("</LF>.*", "", "g", k)
            w = gensub("</LF>.*", "", "g", q0)
            z = sub(".*<LF>", "", u)
            z = sub(".*<LF>", "", v)
            z = sub(".*<LF>", "", w)
            z = gsub("[^a-zA-Z0-9]", "", u)
            z = gsub("[^a-zA-Z0-9]", "", v)
            z = gsub("[^a-zA-Z0-9]", "", w)
            x = length(u)
            y = length(v)
            z = x < y ? x : y
            x = 1
            while (x <= z && index(substr(u, 1, x), substr(v, 1, x)) == 1) {
              x = x + 1
            }
            z = length(w)
            z = z < y ? z : y
            y = 1
            while (y <= z && index(substr(w, 1, y), substr(v, 1, y)) == 1) {
              y = y + 1
            }
          }
        }
        if (x < y) {        #matched up to following item
#print "::nxt::", substr(u, 1, x), substr(v, 1, y), substr(w, 1, y)
          k = k q0 RS
```

```
          n = n "|" q2
        } else {        #matched up to previous item
#print "::prv::", substr(u, 1, x), substr(v, 1, x), substr(w, 1, y)
          j = j k
          m = m "|" n
          k = q0 RS
          n = q2
        }
      }
    }
  } else {          #no head word, continue
#print "::!LF::"
    k = k q0 RS
    n = n "|" q2
  }
}
```

B.2 Nexus Glossarization

This script takes the output of the previous script and constructs a glossary of all of the terms contained in it, and lists of all of the words used to define those terms. All of the rules used by the script appear with the code that uses them, for clarity. The ruleset is separable from the engine that executes them, but it is simpler to present them together.

```
import sys, regex, regsub, string

# Newly modified:  7/5/00
# Usage:  Biggerweb.py <WEBfile(s)> <wordfile.txt>
# this script grabs dictionary headwords and associates them with definitions

# global set of tag patterns used to parse the dictionary
#  Very few changes for OED patterns
hw = '<LF>'  # <HL>?
wf = '<LF>'  # <CF> <XL>?
asp = '<LF>'  # <VF>
plw = '<LF>'
singw = '<LF>'
```

```
wordsep = hw
pattern1 = regex.compile(hw)
bdef = '<DEF>'
pattern2 = regex.compile(bdef)
ehw = '</LF>'
pat_end1 = regex.compile(ehw)
edef = '</DEF>'
pat_end2 = regex.compile(edef)
ewf = '</SF>'
pat_end3 = regex.compile(ewf)
easp = '</LF>'
pat_end4 = regex.compile(easp)
eplw = '</LF>'
pat_end5 = regex.compile(eplw)
esingw = '</LF>'
pat_end6 = regex.compile(esingw)

acc_sign = "&[^\.][^\.]*." # pattern for accented character error checking
acc_patt = regex.compile(acc_sign)

xpt_sign = "[)!?-]\|  \|^ \|[, :] *$" # pattern for special char. error checking
xpt_patt = regex.compile(xpt_sign)

spl_sign = "^[28]" # spelling error checking (first letters)
spl_patt = regex.compile(spl_sign)

def_sign = "&[^\.][^\.]*.\|--\|[\]'\|" '[<"]' # pattern for def. error checking
def_patt = regex.compile(def_sign)

xsw_sign = "[   ]" # pattern for erroneous single word definitions
xsw_patt = regex.compile(xsw_sign)


# returns in a single line an entry with at least 1 head word and 1 definition
# the functionality of this code is implemented elsewhere in AWK for speed
def getchunk (chunk):
  nextl = input.readline()
```

```
  return [nextl, chunk]


# returns head words removing special characters and spelling errors
def fixwords (d):
  y = 0
  for z in d:
    zy = regsub.sub(ehw+'.*', '', z)
    if len(zy) == 0:  # works since SF is after LF in text
      zy = regsub.sub(ewf+'.*', '', z)
      zy = regsub.sub('.*<SF>', '', zy)
    d[y] = zy
    y = y + 1
  accents(d)
  excepts(d)
  if len(d) > 1:    # remove ones that don't match properly
    yyy = string.lower(regsub.sub(' .*', '', d[0]))
    ly = len(yyy) / 3 + 2
    lyy = len(yyy) * 2 / 3
    zz = regex.compile(yyy[:ly])
    zzz = regex.compile(' ' + yyy[:lyy])
    y = 1
    for z in d[y:]:
      z = string.lower(z)
      if zz.match(z) < 1 and (lyy < 4 or zzz.search(z) < 0):
        del d[y]
      else:
        y = y + 1
  addhead(d)

  return d


# checks if a head word has been previously defined
def chkword (b, z):
  if b.has_key(z):
    tmp = b[z]
```

```
  else:
    z = string.capitalize(z)
    if b.has_key(z):
      tmp = b[z]
    else:
      z = string.capwords(z)
      if b.has_key(z):
        tmp = b[z]
      else:
        z = string.lower(z)
        if b.has_key(z):
          tmp = b[z]
        else:
          tmp = None
  return tmp


# checks if any of the head words have been previously defined
# immediately merges duplicate chunks due to multiple alternate spellings
# happens when at least three chunks logically belong together (~170 cases)
# final list of node numbers is not sequential without further massaging
def chkwords (b, d):
  if len(d) == 1:
    found = chkword(b, d[0])
  else:
    found = None
    fz = ''
    for z in d[:]:
      if not (z[0] == '-'): # usu. strange plural suffixes
        tmp = chkword(b, z)
        if tmp:
          if found and found[0] != tmp[0]:
            print "Matching definition:  %s %d, %s %d" % (fz, found[0], z, tmp[0])
            if found[0] < tmp[0]:
              tmp[0] = found[0]
            else:
              found[0] = tmp[0]
```

```
        found = tmp
        fz = z
  return found


# writes out chunk information along with the words defined in the chunk
def writechk (noerror, w, y, tk):
  if noerror:
    chkout.write('%s %s\n' % (y, tk)) # to def chunk counter file
    noerror = (pat_end2.search(w) >= 0)
    if not noerror:
      print 'No matching end def. tag:', y
  else:
    print 'BAD CHUNK:', w
  return noerror


# writes out definitions word by word followed by their key value and
# chunk number
def writedef (webdef, y, token, chunk):
  noerror = 1
  c = chunk
  for w in webdef[:]:
    endval = writechk(noerror, w, y, token)
    token = ''
    if noerror:
      noerror = endval
      #print '::1::', w
      if endval > 0:
        w = regsub.sub(edef+'.*', '', w)
      #print '::2::', w
      w = fixdefs(w)
      #print '::3::', w
      adef = string.split(w)
      for v in adef[:]:
        webout.write('%s %s %s\n' % (v, y, c)) # to def files
    else:
```

```
      noerror = 1
    c = c + 1
  return c


# returns the string that matched against the rule
def gettrcstr (s, n):
  if len(s) <= 40:
    return s
  else:
    if n > 10:
      b = n - 10
    else:
      b = 0
    if b + 40 > len(s):
      f = len(s)
    else:
      f = b + 40
    return s[b:f]


# handles the tracing of string matching rules
def tracerule (ruleset, s):
  for r in ruleset[:]:
    n = r[-1].search(s)
    if n >= 0: # the rule fires
      r[2] = gettrcstr(s, n)
      r[3] = r[3] + 1 # r[3] counts the number of firings
      s = regsub.gsub(r[0], r[1], s) # may multiply fire
  return s


usedexpt = []

# handles the tracing of exception matching rules
def excptrule (ruleset, s):
  y = 0
```

```
  for r in ruleset[:]:
    n = r[-1].search(s)
    if r[3] < 0 and n >= 0: # the rule fires
      r[2] = gettrcstr(s, n)
      r[3] = r[3] + 1 # -r[3] counts times the rule may fire
      s = regsub.sub(r[0], r[1], s) # handle unique exception

    if r[3] == 0: # the rule has been used up
      usedexpt.append(r)
      del ruleset[y]
    else: # go on to the next rule
      y = y + 1
  return s


# handles making regular expressions objects out of the re strings
def makere (r):
  o = regex.compile(r[0])
  r.append(o)


# handles the printing out of the traces
def showtrace (r):
  global rulecount
  p1 = 'Rule #' + str(rulecount) + ' '
  rulecount = rulecount + 1
  p3 = ':  [' + r[0] + '] --> [' + r[1] + '] '
  if r[3] > 1:
    trcout.write('%sfired %d times%s\n' % (p1, r[3], p3))
  elif r[3] < 0:
    trcout.write('%shas %d firings left%s\n' % (p1, -r[3], p3))
  elif r[2] != '':
    trcout.write('%sfired at "%s"%s\n' % (p1, r[2], p3))
  else:
    trcout.write('%sdid not fire%s\n' % (p1, p3))
```

```
accrule = [    # few special characters in head words
  ["&asg.", 'g', '', 0],  # not sure here
  ["&edh.", 'th', '', 0],
  ["&th.", 'th', '', 0],
  ["^&ygh.", 'y', '', 0], # old yogh sound
  [" &ygh.", 'y', '', 0],
  ["&ygh.", 'gh', '', 0],
  ["&amp.", '&', '', 0] ]

accexpt = [    # special definitions
  ["&104.", 'c.', '', -1],
  ["&Pi.", 'Product Symbol', '', -1],
  ["&pi.", 'pi', '', -1],
  ["&psi.&p.", "psi'", '', -1],
  ["&300.", 'resp.', '', -1],
  ["&302.", 'reg.', '', -1],
  ["&491.", 'levo', '', -1], # a chemical symbol?
  ["&pstlg.1", 'pound', '', -1], # pound coin
  ["&pstlg.", 'L', '', -2] ] # British pound symbol


# handles the transformation of special characters to ASCII
def accents (d):
  y = 0
  for z in d:
    # z = tracerule (accrule[0:2], z) # for most common ones
    if acc_patt.search(z) >= 0:
      z = tracerule (accrule, z)
      z = excptrule (accexpt, z)
    d[y] = z
    y = y + 1
#  return d

allexpt = [
  ["[!?]", '', '', -641],       # remove !,? in head words
  ["   *", ' ', '', -300],    # multiple space in head words
  ["^  *", '', '', -280],       # initial space in head words
```

```
[" *[ ,:] *$", '', '', -710],    # final space in head words
[" *- *", '-', '', -3303],    # space in hyphenated head words
[" -[a-z]*, -[a-z].*$", '', '', -45],  # remove , -... in head words
[", *-[2 a-z].*$", '', '', -152],  # remove , -... in head words

["(graul)", "graul", '', -1],
["(-ys)", "-ys", '', -1],
["(ruchche)", "ruchche", '', -1],
["(tiss-)", "tissic", '', -1],
["(Zabiism)", "Zabiism", '', -1],
["joyse)", "joyse", '', -1],
["daw)", "daw", '', -1],
["Riemann zeta ) function", "Riemann zeta function", '', -1],
["Cape) leaping hare", "Cape leaping hare", '', -1],
["la) vie interieure", "vie interieure", '', -1],
["hagh)e", "haghe", '', -1],
["Our) Lady eve, even", "Our Lady eve", '', -1],
["St.) Martin's eve", "St. Martin's eve", '', -1],
["American) bastard sanicle", "American bastard sanicle", '', -1],
["correcting) plate", "correcting plate", '', -1],
["la) vie intime", "vie intime", '', -1],
["East) Indian rosewood", "East Indian rosewood", '', -1],
["earth-)inductor compass", "earth-inductor compass", '', -1],
["St.) Cuthbert's duck", "St. Cuthbert's duck", '', -1],
["Miss) Lonelyhearts", "Miss Lonelyhearts", '', -1],
["fried) potatoes", "fried potatoes", '', -1],
["felt-tipped) pen", "felt-tipped pen", '', -1],
["St.) Cuthbert's beads", "St. Cuthbert's beads", '', -1],
["Our) Lady-psalter", "Our Lady-psalter", '', -1],
["St.) Martin's", "St. Martin's", '', -1],
["fur-) side", "fur-side", '', -1],
["h)ostry faggot", "hostry faggot", '', -1],
["just) for the hell of it", "just for the hell of it", '', -1],
["kit-key)", "kit-key", '', -1],
["Colonel) Blimp", "Colonel Blimp", '', -1],
["free) mason-wasp", "free mason-wasp", '', -1],
["white-)wave", "white-wave", '', -1],
```

```
  ["acute) yellow atrophy", "acute yellow atrophy", '', -1],
  ["old) soldier bird", "old soldier bird", '', -1],
  ["high) cargeing", "high cargeing", '', -1],
  ["Our) Lady's hair", "Our Lady's hair", '', -1],
  ["helicopter) gunship", "helicopter gunship", '', -1],
  ["golden) maidenhair-moss", "golden maidenhair-moss", '', -1],
  ["Dr.) Strangelovean", "strangelovean", '', -1],
  ["Saint) Mary's flower", "Saint Mary's flower", '', -1],
  ["rattlesnake) master weed", "rattlesnake master weed", '', -1],
  ["cross) mitre drain", "cross mitre drain", '', -1],
  ["in), on, upon one's top", "in, on, upon one's top", '', -1],
  ["net) muslin", "net muslin", '', -1],
  ["critical) fusion frequency", "critical fusion frequency", '', -1],
  ["great) crested grebe", "great crested grebe", '', -1],
  ["Lord) Steward of the King's Household", \
"Steward of the King's Household", '', -1],
  ["Pitot's) tube", "Pitot's tube", '', -1],
  ["squash-melon) pumpkin", "squash-melon pumpkin", '', -1],
  ["the) time was been, shall be", "the time was been, shall be", '', -1],
  ["weather-beaten-bet)", "weather-beaten-bet", '', -1],
  ["Pyrenean or guard) dog", "Pyrenean dog", '', -1],
  ["Onsager or reciprocity) relation", "Onsager relation", '', -1],
  ["in statu quo prius, or nunc)", "in statu quo prius", '', -1],
  ["put thing) in or into person's) head", "put in a person's head", '', -1],
  ["Hefner or amyl-acetate) lamp", "Hefner lamp", '', -1],
  ["chilli chile or chili) con carne", "chile con carne", '', -1],
  ["high solemn or great) mass", "high solemn mass", '', -1],
  ["Russian etc.) encephalitis", "Russian encephalitis", '', -1],
  ["first- second-, etc.) generation", "first-generation", '', -1],
  ["cry fight, plague, slave, tease, tire, weary, weep, etc.) \
one's heart out", "cry one's heart out", '', -1],
  ["in spite of maugre, etc.) one's teeth", "in spite of one's teeth", '', -1],
  ["squatter's squatters'; occas. squatter) right", "squatter's right", '', -1],
  ["on a commonly the) wind", "on the wind", '', -1],
  ["on upon) the, or one's, way", "upon the way", '', -1],
  ["young Young) America", "Young America", '', -1],
  ["turn the one's) back", "turn one's back", '', -1],
```

```
  ["WW II)", "WW II", '', -1],
  ["to work do obs.) annoy", "to work do", '', -1],
  ["what-ye -you) -calletc.)", "what-you-call", '', -1],

  [" [^ ][^ ]*)", '', '', -106]]    # remove ) in head words

splexpt = [
  ["8", "section eight", '', -1],
  ["2,4,5-T", "2-T", '', -1]]

# handles spelling errors and other bizarre stuff in the head words
def excepts (d):
  y = 0
  for z in d[:]:
    if xpt_patt.search(z) >= 0: # non-alphabetic "[?.(0-9/<>^\]"
      z = excptrule (allexpt, z)
      if len(z) > 0:
        d[y] = z
        y = y + 1
      else:
        print 'Deleted word:  ' + d[y]
        del d[y]

    elif spl_patt.search(z) == 0: # match "^[abce-npr-uwA-LNOPR-W]"
      z = excptrule (splexpt, z)
      if len(z) > 0:
        d[y] = z
        y = y + 1
      else:
        print 'Deleted word:  ' + d[y]
        del d[y]
    else:
      y = y + 1
#  return d

headexpt = [ # all fire exactly once (to add second word to node)
  ["^aconitia", 'aconitic', '', -1],
```

```
["^anguine lizard", 'lizard', '', -1],
["^anise hyssop", 'hyssop', '', -1],
["^Arkansas cabbage", 'cabbage', '', -1],
["^aronia, thorn", 'thorn', '', -1],
["^black i$", 'ipecac', '', -1],
["^bleinerite", 'bieberite', '', -1],    # least damage
["^Brazil or Brazilian tea", 'tea', '', -1],
["^broom-bush", 'bush', '', -1],
["^brush-grass", 'grass', '', -1],
["^bush-grass", 'grass', '', -1],
["^campholic acid", 'camphol', '', -1],
["^Canadian or Quebec m", 'marmot', '', -1],
["^carbanilide", 'carbanil', '', -1],
["^cat-footedness", 'cat-footed', '', -1],
["^chalice-moss", 'moss', '', -1],
["^close about", 'close in', '', -1],
["^copahuvic", 'copaiva', '', -1],
["^crab-eating seal", 'seal', '', -1],
["^crew cut", 'hair-cut', '', -1],
["^dilettant", 'dilettante', '', -1],
["^ditch-bur", 'burr', '', -1],
["^ditch-grass", 'grass', '', -1],
["^dulcoacid", 'sweet-sour', '', -1],
["^dysgenesic", 'dysgenesis', '', -1],
["^enzymatically", 'enzymatic', '', -1],
["^fairies'-hair", 'fairy-weed', '', -1],
["^false or summer heliotrope", 'heliotrope', '', -1],
["^field-kale", 'kale', '', -1],
["^fleyedly", 'fley', '', -1],
["^foppasty", 'fop', '', -1],
["^fore-goodsire", 'great-grandfather', '', -1],
["^French honey-suckle", 'honey-suck', '', -1],
["^fuller's teazel", 'teazel', '', -1],
["^fuller's thorn", 'thorn', '', -1],
["^gold-breasted trumpeter", 'trumpeter swan', '', -1],
["^gonosphaerium", 'gonosphere', '', -1],
["^G\.T\.T\.", 'gone', '', -1],
```

```
["^gum-thistle", 'thistle', '', -1],
["^handgriping", 'handgrip', '', -1],
["^hare's-bane", 'bane', '', -1],
["^hisis", 'his', '', -1],
["^hithertoward", 'hitherto', '', -1],
["^horse-willow", 'willow', '', -1],
["^hydantoin", 'hydantoic', '', -1],
["^incense c$", 'cedar', '', -1],
["^inexpleble", 'inexpleable', '', -1],
["^Irish or St\. Dabeoc's h", 'heather', '', -1],
["^king-penguin", 'penguin', '', -1],
["^knot-grass moth", 'moth', '', -1],
["^kolbasa", 'kielbasa', '', -1],
["^ligno-sulphuric", 'lignosulphonic acid', '', -1],
["^lock-knit", 'Locknit', '', -1],
["^lotus-berry", 'berry', '', -1],
["^louse-burr", 'burr', '', -1],
["^Malay porcupine", 'porcupine', '', -1],
["^manganicyanhydric acid", 'manganicyanide', '', -1],
["^marsh fern", 'fern', '', -1],
["^Mississippi k", 'kite', '', -1],
["^monkey guava", 'guava', '', -1],
["^mountain s$", 'sweetwood', '', -1],
["^musk-scabious", 'musk', '', -1],
["^narcoticalness", 'narcotical', '', -1],
["^neurilematic", 'neurilemma', '', -1],
["^obedientiar", 'obedientiary', '', -1],
["^oil of cedar", 'cedar', '', -1],
["^overmodiness", 'overmod', '', -1],
["^pin-setting", 'pin-setter', '', -1],
["^potash-felspar", 'feldspar', '', -1],
["^pounce-tree", 'tree', '', -1],
["^pseudo-bacillus", 'bacillus', '', -1],
["^pseudo-bacterium", 'bacterium', '', -1],
["^Ptolemaean", 'Ptolemaian', '', -1],
["^pyrocamphretic acid", 'pyro-camphretic', '', -1],
["^pyromellitic acid", 'mellitic acid', '', -1],
```

```
["^pyrophosphamic acid", 'pyrophosphamic', '', -1],
["^quack-grass", 'grass', '', -1],
["^Queensland box", 'red box', '', -1],
["^quinquertian", 'pentathletical', '', -1],
["^r\. night-jar", 'nightjar', '', -1],
["^rebreathing", 'rebreathe', '', -1],
["^red m$", 'red manganese', '', -1],
["^reforestization", 'reforestation', '', -1],
["^renovater", 'renovate', '', -1],
["^retrogradingly", 'retrogradely', '', -1],
["^ringtail hawk", 'hawk', '', -1],
["^rope's-ending", "rope's-end", '', -1],
["^round h", 'herring', '', -1],
["^sable-mouse", 'mouse', '', -1],
["^Saint Mary's seed", 'seed', '', -1],
["^satin bower-bird", 'bower-bird', '', -1],
["^scarlet mite", 'mite', '', -1],
["^serpently", 'serpent-like', '', -1],
["^sexto-decimo", 'sextodecimo', '', -1],
["^shrubby tansy", 'tansy', '', -1],
["^slim-down", 'slim', '', -1],
["^smarty-boots", 'smarty-pants', '', -1],
["^standing c$", 'cypress', '', -1],
["^sweet bent", 'bent', '', -1],
["^tetradecinene", 'tetradecenyl', '', -1],
["^the whole world", 'the world', '', -1],
["^thoraco-centesis", 'thoracocentesis', '', -1],
["^thymotide", 'thymotic', '', -1],
["^tin p", 'tin-pyrites', '', -1],
["^track system", 'tenure track', '', -1],
["^trey-bit", 'tray-bit', '', -1],
["^triethylurea", 'triethylmethane', '', -1],
["^trimethyl phosphate", 'trimethyl-phosphine', '', -1],
["^unshakeable", 'unshakable', '', -1],
["^water trefoil", 'trefoil', '', -1],
["^water-stoma", 'stoma', '', -1],
["^white chameleon", 'chameleon', '', -1],
```

```
  ["^wide-body", 'wide-bodied', '', -1],

  ["^and$", '&', '', -1],
  ["^learn$", 'learnt', '', -1],
  ["^sleep$", 'slept', '', -1],
  ["^antenna", 'antennae', '', -1],
  ["^cilia", 'cilium', '', -1],
  ["^formula", 'formulae', '', -1],
  ["^fungus", 'fungi', '', -1],
  ["^focus", 'foci', '', -1],
  ["^locus", 'loci', '', -1],
  ["^nebula", 'nebulae', '', -1],
  ["^craftsman", 'craftsmen', '', -1],
  ["^draughtsman", 'draughtsmen', '', -1],
  ["^huntsman", 'huntsmen', '', -1],
  ["^radius", 'radii', '', -1],
  ["^sportsman", 'sportsmen', '', -1],
  ["^stimulus", 'stimuli', '', -1] ]

#  ["^And", '&', '', -1],


# handles addition of abbreviations or irregular forms in head word list
def addhead (d):
#  global headexpt
  x = -1
  result = []
  for z in d[:]:
    y = 0
    for head in headexpt[:]:
      if x < 0 and head[-1].search(z) == 0:
        head[2] = z
        result = head
        x = y
      y = y + 1
  if x >= 0: # only one ever fires at a time
    d.append(result[1])
```

```
    result[3] = result[3] + 1 # -result[3] is times the rule fires
    usedexpt.append(result)
    del headexpt[x]
#  return d


defsrule = [    # fewer patterns to remove in the OED
  ['[][(),;:!?"*`]', '', '', 0], # not ok (1st with punct., 2nd without)
  ["<QP>.*</QP>", '', '', 0],  # quote paragraph (fixes to source)
  ["<#>.*</#>", '', '', 0],  # quote paragraph (fixes to source)
  ["</?S[24678]>", '', '', 0],
  ["<[IB]L>[^L]*<LF>", '', '', 0],  # not exact
  ["</LF>[^L]*</[IB]L>", '', '', 0],  # not exact
  ["<MPR>[^M]*</MPR>", '', '', 0],  # modern pronounciation
  ["<IPR>[^R]*</IPR>", '', '', 0],  # pronounciation
  ["<ST>.[^>]*</ST>", '', '', 0],    # status (before IPH)
  ["<X?DAT>[^>]*</X?DAT>", '', '', 0],  # date
  ["<RX>[^>]*</RX>", '', '', 0],    # cross reference
  ["<FS>[^F]*</FS>", '', '', 0],    # foreign sense ???
  ["<FS>[^<][^<]*$", '', '', 0],    # foreign sense ???
  ["<PS>.[^P]*</PS>", '', '', 0],    # part of speech
  #["<OLD#>[^>]*</OLD#>", '', '', 0],  # pronounciation
  ["<SN>.[^N]*</SN>", '', '', 0],    # sense number
  ["<HO>.[^>]*</HO>", '', '', 0],    # homonym number
  ["<L>.[^>]*</L>", '', '', 0],    # language
  ["<[/MVCS][/<>LFMVCS]*F>", '', '', 0],  # remove consecutive form tags
          # must be last of 'CAP' rules
  ["<fr>.[^f]*</fr>", '', '', 0],    # fraction
  ["<form>.[^f]*</form>", '', '', 0],  # formula
  ["<just>.[^f]*</just>", '', '', 0],  # formula
  ["<chem>.[^f]*</chem>", '', '', 0],  # chemical formula
  ["<gk>.[^>]*</gk>", '', '', 0],    # greek
  ["<sc>.[^>]*</sc>", '', '', 0],
  ["<su>.[^>]*</su>", '', '', 0],
  ["</?XL>", '', '', 0],
  ["</?XR>", '', '', 0],
  ["</?XIL>", '', '', 0],
```

```
["</?IL>", '', '', 0],
["</?LB>", '', '', 0],
["</?ETN>", '', '', 0],
["</?col>", '', '', 0],
["</?ln>", '', '', 0],
["</?s[suc]>", '', '', 0],
["</?ss[bi]>", '', '', 0],
["</?r>", '', '', 0],
["</?b>", '', '', 0],
["</?i>", '', '', 0],
["</?in>", '', '', 0],
["</?p>", '', '', 0],
["</?note>", '', '', 0],
["&[1-9][0-9][0-9].", '', '', 0],
["&74.", '', '', 0],
["&amp.", '&', '', 0],
["&Oe.", 'Oe', '', 0],
["&oe.", 'oe', '', 0],
["&Ae.", 'Ae', '', 0],
["&ae.", 'ae', '', 0],
["&ccdil.", 'c', '', 0],
["&Aacu.", 'A', '', 0],
["&aacu.", 'a', '', 0],
["&Eacu.", 'E', '', 0],
["&eacu.", 'e', '', 0],
["&iacu.", 'i', '', 0],
["&Oacu.", 'O', '', 0],
["&oacu.", 'o', '', 0],
["&uacu.", 'u', '', 0],
["&egrave.", 'e', '', 0],
["&igrave.", 'i', '', 0],
["&ugrave.", 'u', '', 0],
["&acirc.", 'a', '', 0],
["&Ecirc.", 'E', '', 0],
["&ecirc.", 'e', '', 0],
["&icirc.", 'i', '', 0],
["&ocirc.", 'o', '', 0],
```

```
  ["&ucirc.", 'u', '', 0],
  ["&Auml.", 'A', '', 0],
  ["&auml.", 'a', '', 0],
  ["&euml.", 'e', '', 0],
  ["&iuml.", 'i', '', 0],
  ["&Ouml.", 'O', '', 0],
  ["&ouml.", 'o', '', 0],
  ["&Uuml.", 'U', '', 0],
  ["&uuml.", 'u', '', 0],
  ["&obar.", 'o', '', 0],
  ["&Th.", 'Th', '', 0],
  ["&th.", 'th', '', 0],
  ["&Edh.", 'Th', '', 0],
  ["&edh.", 'th', '', 0],
  ["&ygh.", 'gh', '', 0],
  ["&Ygh.", 'Y', '', 0],
  ["&pstlg.", '', '', 0],    # pound symbol
  ["&sm.", '', '', 0],
  ["&sd.", '', '', 0],
  ["&acu.", '', '', 0],
  ["&breve.", '', '', 0],
  ["&lenis.", '', '', 0],
  ["&mac.", '', '', 0],
  ["&dubh.", '', '', 0],
  ["&dag.", '', '', 0],
  ["&deg.", '', '', 0],
  ["&para.", '', '', 0],
  ["&times.", '', '', 0],
  ["<or/", 'or', '', 0] ]

defsexpt = [ # given in order they appear (only for single word definitions)
  ['Apostacy', 'apostasy', '', -1] ]


# handles the removal of special characters in the definitions
def fixdefs (z):
  z = tracerule (defsrule[0:1], z)
```

```
  if def_patt.search(z) >= 0: # key patterns in defs "--\|[\\]'\|<"
    z = tracerule (defsrule[1:], z)
  elif xsw_patt.search(z) < 0: # only single word exceptions "[^ ]<"
    z = excptrule (defsexpt, z)
  return z


# does the code run
def maincode (args):
  global input, zout, webout, chkout, trcout, rulecount
  map(makere, accrule)
  map(makere, accexpt)
  map(makere, allexpt)
  map(makere, splexpt)
  map(makere, headexpt)
  map(makere, defsrule)
  map(makere, defsexpt)
  rhw = []
  rdf = []
  curr = 0
  a = {}
  t = []
  bins = []
  chkstr = regsub.sub('\.txt', '', args[-1]) + '.chk'
  print chkstr # trace output files
  for z in args[:-1]:
    print z # dictionary input files
    input = open(z, 'r')
    zout = regsub.sub('\.txt', '', z) + '.def'
    cbin = (zout, [], [], [])
    line = input.readline()
    while line:
      #n = pattern1.search(line)
      #if n >= 0:
        # ONLY the lookup form is considered in OED
        t = ['', line[:-1]]
        #t = getchunk(line[n:-1])
```

```
        #line = t[0]
        #c = fixwords(regsub.split(t[1], wordsep)[1:2])
        #c = fixwords(regsub.split(t[1], wordsep)[1:])
        c = fixwords(string.split(t[1], wordsep)[1:])
        rhw.append(c)
        rdf.append(t[1])
        v = chkwords(a, c)
        if not v:
          curr = curr + 1
          v = [curr]
        tk = []
        for token in c[:]:
          if not a.has_key(token):
            a[token] = v
            tk.append(token)
        cbin[1].append(t[1])
        cbin[2].append(v)
        cbin[3].append(tk)
      #else:
        line = input.readline()
    input.close()
    bins.append(cbin)
  chkout = open(chkstr, 'w')
  aa = {}
  cc = {}
  n = 0
  for z in bins:
    webout = open(z[0], 'w')
    print z[0]  # definition output files
    lenz = len(z[1])
    i = 0
    while i < lenz:
      zz = regsub.split(z[1][i], bdef)[1:]
      rr = z[3][i]
      v = z[2][i][0]
      if not aa.has_key(v):
        n = n + 1
```

```
      aa[v] = n
      nn = 1
      cc[n] = nn
    elif len(rr) == 0:
      nn = cc[aa[v]]
    else:
      nn = 1
    vv = aa[v]
    nn = writedef(zz, vv, rr, nn)
    if nn > cc[aa[v]]:
      cc[aa[v]] = nn
    i = i + 1
  webout.close()
chkout.close()

print args[-1]  # head word output file
output = open(args[-1], 'w')
i = 0
tenth = (len(a.items()) + 9) / 10
for w in a.items():
  output.write('%s %s\n' % (w[0], aa[w[1][0]]))
  i = i + 1
  if i % tenth == 0:
    print i
output.close()

trcstr = regsub.sub('\.txt', '', args[-1]) + '.trc'
print trcstr # trace output files
trcout = open(trcstr, 'w')
rulecount = 0
map(showtrace, accrule)
map(showtrace, accexpt)
map(showtrace, allexpt)
map(showtrace, splexpt)
map(showtrace, headexpt)
map(showtrace, defsrule)
map(showtrace, defsexpt)
```

```
  map(showtrace, usedexpt)
  trcout.close()

        defstr = regsub.sub('\.txt', '', args[-1]) + '.raw'
        print defstr # raw definition output files
  defout = open(defstr, 'w')
  i = 0
  for ri in rhw:
    rdfs = '%s' % (aa[a[ri[0]][0]],)
    for rj in ri:
      rdfs = rdfs + ' ' + rj
    defout.write('%s\n%s\n\n' % (rdfs, rdf[i]))
    i = i + 1
  defout.close()


input = None
zout = None
webout = None
chkout = None
trcout = None
rulecount = 0

if len(sys.argv) > 1:
  maincode(sys.argv[1:])
```

B.3 Nexus Extraction

The output of the glossarization step feeds into the nexus extractor, which generates a list of all of the arcs in the nexus, as well as a list giving the most popular terms for each definition (in general, there are more than one term per definition). Again all of the morphological rules used in this script accompany the code that executes them.

```
import sys, regex, regsub, string

# Newly modified:  4/25/99
# Usage:  Graph.py wordfile.txt <WEBfile(s).*.def> graphfile.txt
```

```
# this script creates the graph of the dictionary words

a = {}
aa = {}
aaa = {}
exlines = []
grout = open(sys.argv[-1], 'w')


# returns a line with a FROM node number and a TO node number and a counter
def wordarc (org, dest, wc):
  aa[dest] = aa[dest] + 1
  if aa[dest] > aa[aaa[a[dest]]]:
    aaa[a[dest]] = dest
  return "%s %s %d\n" % (org, a[dest], wc)


# stemming suffixes
sfdc = 'bb$\|ch$\|dd$\|ff$\|gg$\|kk$\|ll$\|mm$\|nn$\|pp$\|rr$\|s[hs]$\|tt$\|zz$'
rxdc = regex.compile(sfdc)
sfly = 'ly$'; rxly = regex.compile(sfly)
sfing = 'ing$'; rxing = regex.compile(sfing)
sfdrs = '[drs]$'
sfedr = 'e' + sfdrs; rxedr = regex.compile(sfedr)
sfiedr = 'ie' + sfdrs; rxiedr = regex.compile(sfiedr)
sfst = 'est$'; rxst = regex.compile(sfst)
sfist = 'i' + sfst; rxist = regex.compile(sfist)
sfaps = "'?s$"; rxaps = regex.compile(sfaps)
sflike = 'like$'; rxlike = regex.compile(sflike)
sfn = 'n$'; rxewn = regex.compile('[ew]' + sfn)
sfen = 'e' + sfn
sften = 't' + sfen; rxten = regex.compile(sften)
rxtten = regex.compile('t' + sften)
sfgtn = 'ot' + sften; rxgtn = regex.compile('g' + sfgtn)
rxkvn = regex.compile('[kv]en$')
rxaown = regex.compile('[nrs][ao]wn$')
rxhst = regex.compile('[hst]$')
```

```
# returns an arc specified by stemming the word in the definition file
#  handles most stemming problems
def stemarc (wc, arc, key):
  arcline = ""
  plok = 1 # try removing s from stem
  trye = 0 # don't try adding e to stem
  dblc = 0 # don't check for double consonant
  if rxly.search(arc) > 0:
    arc = regsub.sub(sfly, '', arc)
    plok = 0
  if rxing.search(arc) > 0:
    trye = 1
    dblc = 1
    arc = regsub.sub(sfing, '', arc)
  elif rxedr.search(arc) > 0:
    if rxiedr.search(arc) > 0:
      if len(arc) > 4:
        arc = regsub.sub(sfiedr, 'y', arc)
      else:
        arc = regsub.sub(sfdrs, '', arc)
    else:
      dblc = 1
      trye = 1
      arc = regsub.sub(sfedr, '', arc)
  elif plok and rxaps.search(arc) > 0:
    arc = regsub.sub(sfaps, '', arc)
  elif rxst.search(arc) > 0:
    if rxist.search(arc) > 0:
      arc = regsub.sub(sfist, 'y', arc)
    else:
      dblc = 1
      trye = 1
      arc = regsub.sub(sfst, '', arc)
  elif rxlike.search(arc) > 0:
    arc = regsub.sub(sflike, '', arc)
```

```
  elif rxewn.search(arc) > 0:
    if rxten.search(arc) > 0:
      if rxtten.search(arc) > 0:
        if rxgtn.search(arc) > 0:
          arc = regsub.sub(sfgtn, 'et', arc)
        else:
          arc = regsub.sub(sften, 'e', arc)
      else:
        arc = regsub.sub(sfen, '', arc)
    elif rxkvn.search(arc) > 0 or rxaown.search(arc) > 0:
      arc = regsub.sub(sfn, '', arc)

  if dblc and (rxdc.search(arc) > 0): # ff, ss don't stem
    if rxhst.search(arc) > 0:
      trye = -1
      arce = arc + 'e'
    else:
      trye = 0
    a2 = regsub.sub('tt$', 't', arc) # no good for boycott
    a2 = regsub.sub('pp$', 'p', a2)
    a2 = regsub.sub('gg$', 'g', a2)
    a2 = regsub.sub('nn$', 'n', a2)
    a2 = regsub.sub('mm$', 'm', a2)
    a2 = regsub.sub('kk$', 'k', a2)
    if len(arc) > 3:
      a2 = regsub.sub('bb$', 'b', a2) # ebb
      a2 = regsub.sub('dd$', 'd', a2) # add
      a2 = regsub.sub('rr$', 'r', a2) # err
      if len(arc) > 4:
# no good for frizz
        a2 = regsub.sub('zz$', 'z', a2)
        if len(arc) > 5:
# no good for foretell, recall, befall, stroll, thrill, shrill, appall
          a2 = regsub.sub('ll$', 'l', a2)
    if a2 != arc:
      arc = a2
```

```
  if trye > 0:
    arce = arc + 'e'

  if trye > 0 and a.has_key(arce):
    arcline = wordarc (key, arce, wc)
  elif a.has_key(arc):
    arcline = wordarc (key, arc, wc)
  elif trye < 0 and a.has_key(arce):
    arcline = wordarc (key, arce, wc)

  return arcline


# handles common place name suffixes and latin terms
def placearc (wc, arc, key):
  global mkey
  arcline = ""
  t1 = not a.has_key(string.lower(arc))
  sarc = string.lower(arc)
  arc = regsub.sub("[.']*$\|[']s$", '', arc)
  nosarc = string.lower(regsub.sub("s$", '', arc))
  tst = not (a.has_key(sarc) or a.has_key(string.lower(arc)) or \
             a.has_key(nosarc)) and len(nosarc) > 3
  if tst and regex.search('^[a-zA-Z].*[a-z]$', arc) >= 0:
    oldarc = arc
    arc = regsub.sub("n$", 'nian', arc)
    arc = regsub.sub("a$", 'an', arc)
    arc = regsub.sub("e$", 'ean', arc)
    arc = regsub.sub("i$", 'ian', arc)
    arc = regsub.sub("iensis$", 'ian', arc)
    arc = regsub.sub("is$", 'isian', arc)
    arc = regsub.sub("[os]$", 'an', arc)
    arc = regsub.sub("gium$", 'gian', arc)
    arc = regsub.sub("way$", 'wegian', arc)
    arc = regsub.sub("y$", 'ian', arc)
    if regex.search('land$', arc) >= 0:
      arc = regsub.sub("ngland", 'nglish', arc)
```

```
      arc = regsub.sub("reland", 'rish', arc)
      arc = regsub.sub("otland", 'ottish', arc)
      arc = regsub.sub("land", 'lander', arc)
    elif regex.search("Denmark$\|Portugal$\|Swedenian$\|\
Spainian$\|Germanian$\|Francean$", arc) >= 0:
      arc = regsub.sub("enmark", 'anish', arc)
      arc = regsub.sub("gal", 'guese', arc)
      arc = regsub.sub("edenian", 'edish', arc)
      arc = regsub.sub("painian", 'panish', arc)
      arc = regsub.sub("ancean", 'ench', arc)
      arc = regsub.sub("manian", 'man', arc)
    elif regex.search('[^n]$', arc) >= 0:
      arc = regsub.sub("$", 'ian', arc)

    arc = string.lower(arc)
    larc = string.lower(oldarc)
    if (regex.search('^[A-Z]', oldarc) == 0) and a.has_key(arc):
      a[larc] = mkey
      aa[larc] = 0
      aaa[mkey] = larc
      wordout.write('%s %s\n' % (oldarc, mkey))
      arcline = wordarc (key, larc, wc) + wordarc (mkey, arc, 1)
      mkey = mkey + 1
    else:
      arc = larc
      arc = regsub.sub("a$\|us$\|um$\|is$", '', arc)
      if (len(arc) > 6) and (a.has_key(arc)):
        arcline = wordarc (key, arc, wc)
  return arcline


# returns an arc specified by the word in the definition file
def buildarc (wc, arc, key):
  arcline = ""
  if a.has_key(arc):
    arcline = wordarc (key, arc, wc)
  else:
```

```
    arc = regsub.sub("[.']*$\|[']s$", '', arc)
    if a.has_key(arc):
      arcline = wordarc (key, arc, wc)
    else:
      arcline = stemarc (wc, arc, key)
  return arcline


# returns an arc specified by the word in the definition file
def hyphenarc (wc, arc, key):
  arcline = ""
  arch = regsub.sub("[^a-zA-Z]*$", '', arc) + '-'
  if a.has_key(arch):
    arcline = wordarc (key, arch, wc)
  elif a.has_key(arch[:-1]):
    arcline = wordarc (key, arch[:-1], wc)
  return arcline


# finds arcs in simple as well as compound and hyphenated words
def subarc (wc, word, key):
  arcline = ""
  capword = word
  word = string.lower(word)
  if regex.search('[-]', word) < 0:
    arcline = placearc(wc, capword, key)
  if len(arcline) == 0:
    arcline = buildarc(wc, word, key)
  if len(arcline) == 0:
    xline = 0
    hyphens = []
    if regex.search('[-]', word) >= 0:
      word = regsub.gsub('-', ' ', word)
      if a.has_key(word):
        arcline = wordarc (key, word, wc)
      else:
        hyphens = string.split(word)
```

```
        for awd in hyphens:
          aline = hyphenarc(wc, awd, key)
          if len(aline) == 0:
            aline = buildarc(wc, awd, key)
          arcline = arcline + aline
          if len(aline) > 0:
            xline = xline + 1
    lword = len(regsub.sub("[-.']", '', word))
    if len(arcline) > 0:
#      print "hyphens", hyphens, xline
      if xline < len(hyphens):
        grout.write(arcline) # partial success
        arcline = ""
    elif lword > 3:
      s = 2
      while s <= lword - 2:
        arc1 = word[:s]
        arc2 = word[s:]
        s = s + 1
        aline1 = hyphenarc(wc, arc1, key)
        aline2 = buildarc(wc, arc2, key)
        if len(aline1) > 0 and len(aline2) > 0:
          arcline = aline1 + aline2
#          print "compound", arc1 + ' | ' + arc2
          break
  return arcline


# finds arcs in multiword environment
def findarc (wc, w1, w2, w3, key, line):
  found = 0
  if len(w1) > 0:
    word = string.lower(string.join([w1, w2, w3]))
    arcline = buildarc(wc, word, key)
    if len(arcline) == 0:
      word = string.lower(string.join([w1, w2]))
      arcline = buildarc(wc, word, key)
```

```
      if len(arcline) == 0:
        arcline = subarc(wc, w1, key)
        if len(arcline) > 0:
          found = 1
#          print arcline, "one word", (w1, wc)
      else:
        found = 2
#        print arcline, "two wds.", (word, wc)
    else:
      found = 3
#      print arcline, "three wds.", (word, wc)
  elif len(w2) > 0:
    word = string.lower(string.join([w2, w3]))
    arcline = buildarc(wc, word, key)
    if len(arcline) == 0:
      arcline = subarc(wc, w2, key)
      if len(arcline) > 0:
        found = 1
#        print arcline, "penult wd.", (w2, wc)
    else:
      found = 2
#      print arcline, "two last wds.", (word, wc)
  else:
    arcline = subarc(wc, w3, key)
    if len(arcline) > 0:
      found = 1
#      print arcline, "last wd.", (w3, wc)

  if found == 0:
#    print line
    exlines.append("%s %d\n" % (line[:-1], wc))
      # this file never gets too large
  else:
    grout.write(arcline)
  return found
```

```
wordin = open(sys.argv[1], 'r')
wordout = open(regsub.sub('\.txt', '', sys.argv[-1]) + '.sup', 'w')
webfile = wordin.readlines()
wordin.close()
mkey = 0
for line in webfile[:]:
  word = string.lower(regsub.sub(' *[0-9]*$', '', line[:-1]))
  b = string.split(line)
  key = string.atoi(b[-1])
  mkey = max(key, mkey)
  a[word] = key
  aa[word] = 0
  aaa[key] = word

mkey = mkey + 1

wdnum = 1
wdcount = -1
l1 = ""
l2 = ""
l3 = ""
w1 = ""
w2 = ""
w3 = ""
for z in sys.argv[2:-1]:
  print z # dictionary input files
  defsin = open(z, 'r')
  webfile = defsin.readlines()
  defsin.close()

  for line in webfile[:]:    # change to account for chunks
    l1 = l2
    l2 = l3
    l3 = line
    word = regsub.sub(' *[0-9]* ?[0-9]*$', '', line[:-1])
    w1 = w2
    w2 = w3
```

```
    w3 = word
    b = string.split(line)
    key = b[-2]
    chunk = b[-1]    # new model includes chunk number
    r = 0
    if wdnum == key:
      if len(w1) > 0:
        r = findarc(wdcount, w1, w2, w3, wdnum, l1)
      if r > 1:
        w2 = ""
        if r > 2:
          w3 = ""
      wdcount = wdcount + 1
    else:
      if len(w1) > 0:
        r = findarc(wdcount, "", w1, w2, wdnum, l1)
      if r < 2 and len(w2) > 0:
        r = findarc(wdcount + 1, "", "", w2, wdnum, l2)
      w1 = ""
      w2 = ""
      wdnum = key
      wdcount = -1
r = 0
if len(w2) > 0:
  r = findarc(wdcount, "", w2, w3, wdnum, l2)
if r < 2 and len(w3) > 0:
  r = findarc(wdcount + 1, "", "", w3, wdnum, l3)

wordout.close()

exout = open(regsub.sub('\.txt', '', sys.argv[-1]) + '.xpt', 'w')
exout.write(string.joinfields(exlines, ''))
exout.close()

popout = open(regsub.sub('\.txt', '', sys.argv[-1]) + '.pop', 'w')
x = aaa.keys()
x.sort()
```

```
for i in x:
  popout.write('%u %s\n' % (i, string.capwords(aaa[i])))
popout.close()
```

B.4 Nexus Ranking

Once all of the arcs of the graph have been computed, the ranking script computes the ranks of each of the terms based on the arcs of the graph. This is a preliminary step required for the matching computed later on, and corresponds to the implementation of the PageRank algorithm.

```
import sys, regex, regsub, string

# Newly updated:  3/21/99
# Usage:  Qyrank.py inputweb.txt [outputrank] [ranking setup {0.0-1.0,2}]
# this script computes arc adjusted page rank on an input graph file


#  functions to compute NodeRank values
def fullflow(srcl, arcl, val, indx):  # for dampened or undampened model
  srcl[indx] = 0.0
  arcl[indx] = val / a[indx]

#  !!! can withhold smaller and smaller amounts as algorithm converges
strength = 2.0
def ergodic(srcl, arcl, val, indx):  # for ergodic browse model
  srcl[indx] = val / strength
  arcl[indx] = (val - srcl[indx]) / a[indx]

#  !!! can reduce strength as algorithm converges
def selfloop(srcl, arcl, val, indx):  # for structure respecting model
  strength = 2
  arcl[indx] = val / (a[indx] + strength)
  srcl[indx] = strength * arcl[indx]

#  the functions in this list are ranking functions for various models
rankingfuncs = [fullflow, ergodic, selfloop]
```

```
#  compute page rank & sum of ranks
def rankfn(slist1, slist2):
  sum = 0.0
  i = startnode
  for elem in slist2[i:]:
    sum = sum + elem
    tarc[i] = 0.0
    rankingfunctor(slist1, arcv, elem, i)
    i = i + 1
  i = 0
  for line in snod:
    aval[i] = arcv[line]/slist2[dnod[i]]
    tarc[line] = tarc[line] + aval[i]
    i = i + 1
  i = 0
  for line in dnod:
    slist1[line] = slist1[line] + aval[i]/tarc[snod[i]] * \
                   slist2[snod[i]] * (1 - 1.0 / strength)
    i = i + 1
  return sum


#  write out log information
def printlog(iter, sum, dif):
  print "iteration: ", iter, "\tsum: ", initsum, '+', sum - initsum, \
        "\tdiff: ", dif
  wordlog.write("iter: %u\tsum: %g + %12.8g\tdiff: %12.8g\n" \
        % (iter, initsum, sum - initsum, dif))
  wordlog.flush()


#  write out iteration information
def printitr(outfile, ranklist):
  wordout = open(outfile, 'w')
  i = startnode
```

```
  for item in ranklist[i:]:
    wordout.write("%u\t%12.8g\n" % (i, item))
    i = i + 1
  wordout.close()


#  adjust node values after iteration & compute difference between iterations
#  the two functions below are the adjusting functions for the damping
#  and undamping ranking models
def dampen(slist1, slist2):
  diff = 0.0
  i = startnode
  for elem in slist1[i:]:
    slist1[i] = elem * normflow + initflow
    diff = diff + abs(slist1[i] - slist2[i])
    i = i + 1
  return diff


def undamp(slist1, slist2):
  diff = 0.0
  i = startnode
#  fact = initsum / sum
  for elem in slist1[i:]:
#    slist1[i] = elem * fact
    diff = diff + abs(slist1[i] - slist2[i])
    i = i + 1
  return diff


#  handle input and output filename parameters
#  select damping style (can also use command line parameter)
if 1:
  adjustingfunctor = undamp
  rankingfunctor = ergodic  # can be fullflow, ergodic, selfloop
else:
  flow = 0.015625      #steady state flow from graph
```

```
  normflow = 1.0 - flow
  adjustingfunctor = dampen
  rankingfunctor = fullflow  # use only fullflow here
infile = ""
outfile = "pagerk"
logsuff = ".log"
ranksuff = ".i"    # i stands for iteration
argc = len(sys.argv)
if argc < 2 or argc > 4:
  sys.exit(1)
else:
  infile = sys.argv[1]
  if argc > 2:
    outfile = regsub.sub('\.txt', '', sys.argv[2])
  if argc > 3:
    flow = string.atof(sys.argv[3])
    if (flow <= 0.0) or ((flow > 1.0) and (flow != 2)):
      flow = 1.0
    rankfnval = int(flow)
    rankingfunctor = rankingfuncs[rankfnval]
    if (flow - rankfnval) > 0:
      normflow = 1.0 - flow
      adjustingfunctor = dampen
print '%s %s' % (rankingfunctor, adjustingfunctor)
print "reading ", infile
wordin = open(infile, 'r')
inweb = wordin.readlines()
wordin.close()
print "step: ", 1


#  change arc file strings to int array
i = len(inweb)
snod = [0] * i
dnod = [0] * i
aval = [0.0] * i
while i > 0:
```

```
  i = i - 1
  if i % 100000 == 0:
    print "> ", i
  b = string.split(inweb[i])
  snod[i] = string.atoi(b[0])
  dnod[i] = string.atoi(b[1])
  del inweb[i]
print "step: ", 2


#  count number of arcs per node & number of nodes & max mode number
gmax = 0
a = {}
i = len(snod)
while i > 0:
  i = i - 1
  if i % 100000 == 0:
    print "> ", i
  v = snod[i]
  if v > gmax:
    gmax = v
  if a.has_key(v):
    a[v] = a[v] + 1
  else:
    a[v] = 1
print "step: ", 3


# set up the ranking array & verify presence of all nodes
wordlog = open(outfile + logsuff, 'w')
startnode = 1      #not using node #0 here
gnode = len(a)
gsize = gnode + startnode
initsum = 1.0
sum = initsum      #initialize for the adjusting function
diff = sum      #initialize for the log printing function
initial = sum / gnode
```

```
if (adjustingfunctor == dampen):
  initflow = initial * flow       #initialize dampened flow here
ranks = []
ranks.append([0.0] * gsize)
ranks.append([0.0] * startnode + [initial] * gnode)
arcv = [0.0] * gsize
tarc = [0.0] * gsize
i = gmax
while i > 0:
  if not a.has_key(i):
    wordlog.write("empty node %d \n" % (i,))
  i = i - 1
print "step: ", 4


# iterate page rank algorithm
stable = 0
x = 0
while x < 80 and stable < 2:
  sum = rankfn(ranks[x & 1], ranks[x & 1 ^ 1])
  diff = adjustingfunctor(ranks[x & 1], ranks[x & 1 ^ 1])
  printitr(outfile + ranksuff + str(x/10) + str(x%10), ranks[x & 1])
  printlog(x, sum, diff)
  if diff == 0.0:
    stable = stable + 1

  print "step: ", 5 + x
  x = x + 1
wordlog.close()
```

B.5 Nexus Term Matching

Finally, the ArcRank scores are computed for the graph based on the ranks of terms computed in the previous step. The resulting file is the term nexus.

```
import sys, os, re, string, math
```

```
# Newly created:  12/2/99  Redesigned 8/12/00 for OED
# Usage:  Dualink.py OedA.txt qergrk.i100.srt OedA.pop [outprefix]
# this script computes important neighbors to nodes and prepares graph
# output for them.  Memory intensive, but fast


#  functions to compute arc importance value for getallarcs() function below
def dstprp1(src, dst, weight):  # based on proportion of destination value
  return weight / dst

def dstprp2(dst, src, weight):  # based on proportion of destination value
  return weight / dst

#def harmean(src, dst, weight):  # based on harmonic mean
#  weight = weight / 2
#  return weight / src + weight / dst

#def submean(src, dst, weight):  # based on subcontrary harmonic mean
#  return (weight * src + weight * dst) / (src * src + dst * dst)

#def subrmean(src, dst, weight):  # based on subcontrary harmonic mean
#  return (weight * math.sqrt(src) + weight * math.sqrt(dst)) / (src + dst)

#def sub2mean(src, dst, weight):  # based on subcontrary harmonic mean
#  return (weight * src * src + weight * dst * dst) / (src + dst)


#  computes the arc importance for each node
def getallarcs(k, irnk, arcfn):
  j = len(k)
  while j > 0:
    j = j - 1
    l = list(k[j])
    l[1] = arcfn(irnk, ranks[l[0]], l[1])  # compute importance
    l.reverse()
    k[j] = l
  k.sort()
```

```
  q = {}
  v = -1.0
  n = 0
  j = len(k)
  while j > 0:
    j = j - 1
    t = k[j]
    if t[0] != v:
      v = t[0]
      n = n + 1
#      n = p    # n = n + 1 (was the old style)
    q[t[1]] = n
  k.reverse()
  return (k, q)


#  functions to compute arc rank  for rankarcs() function below
def armean(x, y):  # arithmetic mean
  return (x + y) / 2.0

def gmmean(x, y):  # geometric mean
  return math.sqrt(x * y)

def hmmean(x, y):  # harmonic mean
  return x * y * 2.0 / (x + y)

def sbmean(x, y):  # subcontrary harmonic mean
  return (x * x + y * y) / (x + y)


#  computes arc ranks based on arc importance to each endpoint
def rankarcs(i, z, p, q, rankfn):
  k = len(z)
#  if k > 0:
#    zz = 1.0 / k  # so as to normalize by number of arcs
  while k > 0:
    k = k - 1
```

```
    j = z[k][1]
#    qq = 1.0 / len(q[j])  # for normalization purposes
#    z[k][0] = rankfn(p[j] * zz, q[j][i] * qq)
    z[k][0] = rankfn(p[j], q[j][i])    # compute rank
  z.sort()


#  prints graph parameters
#  prints the central node:  (number of arcs) name rank order
def printgraph(i, ival, irnk, of):
  of.write('%s:%-.4g [%u-%u]' % (ival, irnk, len(sv[i]), len(dv[i])))
#  of.write('%s' % (ival,))


#  globals used in printing function
klim = 40
klim2 = klim - 1

#  prints parent/child nodes:  name rank order
#  prints arcs:  strength (order)
def printnodes(i, irnk, z, k, order, of):
  lenz = len(z)
  gw = 0    # gw = 1 needed when gateway taken out
  if order:
    gw = gw + lenz
  if lenz > 0:
    of.write('<>')
    c = 0
    for item in z:    # select items in current row
      j = item[1]
      jrnk = ranks[j]
      if (order ^ (irnk > jrnk)) and (i != j):
        break
      c = c + 1
    if lenz > klim:
      if (c > klim2) and (c < len(z)):
        tt = z[c]
```

```
        z = z[:klim2]
        z.append(tt)
      else:
        z = z[:klim]
    for item in z:
      j = item[1]
      jrnk = ranks[j]
      if order:
        aival = irnk / jrnk / gw
      else:
        aival = jrnk / irnk / (len(sv[j]) + gw)
      of.write('<>%s:%-.4g' % (w[j], aival))
#       of.write('|%s' % (w[j],))


#  fnlist MUST only have one item for the OED
#  list of outputs the script creates and the function that drives the output
fnlist = [  [armean, 'ar']]    # small one for testing
fnlen = len(fnlist)

#  memory is tight so we aggresively delete data that is no longer used
#  handle input and output filename parameters
infile = ""
rkfile = ""
outfile = "out"
outsuff = ".aff"
argc = len(sys.argv)
if argc < 4 or argc > 5:
  sys.exit(1)
else:
  infile = sys.argv[1]
  rkfile = sys.argv[2]
  wdfile = sys.argv[3]
  if argc == 5:
    outfile = sys.argv[4]

print "reading: ", infile
```

```
wordin = open(infile, 'r')  # use OedA.txt
inweb = wordin.readlines()
wordin.close()
del wordin
print "step: ", 1


#  change arc file strings to int array
#  count number of arcs per node & number of nodes & max node number
gmax = 0
a = {}
i = len(inweb)
snod = [0] * i
dnod = [0] * i
while i > 0:
  i = i - 1
  if i % 200000 == 0:
    print "> ", i
  b = string.split(inweb[i])
  del inweb[i]
  snod[i] = string.atoi(b[0])
  dnod[i] = string.atoi(b[1])

  v = snod[i]
  if v > gmax:
    gmax = v
  if a.has_key(v):
    a[v] = a[v] + 1
  else:
    a[v] = 1
# gsize = number of array entries needed assuming nodes are numbered 1, ...
gnode = len(a)
gsize = gnode + 1
del inweb
print "step: ", 2
```

```
#  set up ranks arrays and arc rank arrays
wordin = open(rkfile, 'r')  # use qergrk.i100.srt
inweb = wordin.readlines()
wordin.close()
del wordin
g = 0
j = 0.0
k = 1.0
h = {}
ranks = [0.0] * gsize
sv = [0] * gsize
dv = [0] * gsize
i = len(inweb)
while i > 0:
  i = i - 1
  if i % 50000 == 0:
    print "> ", i
  b = string.split(inweb[i])
  nod = string.atoi(b[0])
  val = string.atof(b[1])
  if not h.has_key(val):
    g = g + 1
    h[val] = g
    j = max(j, val)
    k = min(k, val)
  ranks[nod] = val
  sv[nod] = {}
  dv[nod] = {}
del inweb
del h
print 'there are %d different ranks:  %g is highest, %g is lowest' % (g, j, k)
print "step: ", 3


#  set up word name dictionary
initnum = re.compile('^[0-9]+ ')
wordin = open(wdfile, 'r') # use OedA.pop
```

```
inweb = wordin.readlines()
wordin.close()
del wordin
w = {}
i = len(inweb)
for line in inweb:
  i = i - 1
  if i % 50000 == 0:
    print "> ", i
  u = string.atoi(string.split(line)[0])
  v = initnum.sub('', line[:-1])
  if not w.has_key(u):
    w[u] = v
del inweb
print "step: ", 4


#  compute arc rank array values
i = len(snod)
while i > 0:
  i = i - 1
  if i % 200000 == 0:
    print "> ", i
  u = snod[i]
  del snod[i]
  v = dnod[i]
  del dnod[i]
  arnk = ranks[u] / a[u]
  if sv[u].has_key(v):
    sv[u][v] = sv[u][v] + arnk
  else:
    sv[u][v] = arnk
  if dv[v].has_key(u):
    dv[v][u] = dv[v][u] + arnk
  else:
    dv[v][u] = arnk
del a
```

```
print "step: ", 5


#  open all output files
k = fnlen
while k > 0:
  k = k - 1
  fnlist[k][1] = open(outfile + fnlist[k][1] + outsuff, 'w')


#  compute reverse direction arc importance arrays
sq = [0] * gsize
dq = [0] * gsize
i = 0
while i < gnode:
  if i % 50000 == 0:
    print "> ", i
  i = i + 1
  irnk = ranks[i]
  sq[i] = getallarcs(sv[i].items(), irnk, dstprp1)[1]
  dq[i] = getallarcs(dv[i].items(), irnk, dstprp2)[1]
print "step: ", 6


#  compute forward direction arc importance, arcrank & print out results
i = 0
while i < gnode:
  if i % 50000 == 0:
    print "> ", i
  i = i + 1
  irnk = ranks[i]
  sr = getallarcs(sv[i].items(), irnk, dstprp1)
  dr = getallarcs(dv[i].items(), irnk, dstprp2)
  k = fnlen
  while k > 0:
    k = k - 1
    rankarcs(i, sr[0], sr[1], dq, fnlist[k][0])  #change sr[0]
```

```
    rankarcs(i, dr[0], dr[1], sq, fnlist[k][0])  #change dr[0]
    wordout = fnlist[k][1]
    printgraph(i, w[i], irnk, wordout)    #print node i
    printnodes(i, irnk, sr[0], dr[0], 1, wordout)
    printnodes(i, irnk, dr[0], sr[0], 0, wordout)
    wordout.write('\n')
print "step: ", 7


#  close all output files
for wordout in fnlist:
  wordout[1].close()
print "done.\n"
os._exit(0)
```

B.6 NATO Term Matching

Once the nexus has been computed, we use it as a part of SKEIN to find the matches between pages in the NATO government websites. This code is generic, and can be used to compute matches between any two text inputs. This code's current limiting factor is the time it takes to load the nexus into memory. This loading time accounts for the longest part of the computation, even though the current algorithm has optimized this process to the point where loading speed has improved by two orders of magnitude. Another order of magnitude can easily be gained by inserting the nexus into a database, and by loading only the terms in the nexus specifically related to the terms in the sources.

```
#import os, math, sys, re, string, shelve
import os, math, sys, re, string, marshal

# Newly written:  8/13/00
# Updated:  8/14/00  use cPickle instead of shelve
# Updated:  8/15/00  marshal is by far the fastest
# Usage:  ontocmp.py file1 file2
# this script uses oed dictionary data for onto intersection

sep = '\n'
```

```
dictfile = 'dict.dct'
#dictfile = 'dict.pkl'
#dictfile = 'dict.wds'
#rankfile = 'rank.wds'

datapat = "grep label %s | sed -e 's/.*label=\"//' -e 's/\".*$//'"
comppat = '%s intersecting %s\n'
rankpat = '%s <--%d--> %s'

rulre1 = re.compile('[(,:]')
rulre2 = re.compile("ies ")
rulre3 = re.compile("ies$")
rulre4 = re.compile("'*s ")
rulre5 = re.compile("'*s$")

rep1 = ''
rep2 = 'y '
rep3 = 'y'
rep4 = ' '
rep5 = ''

aaa = open(dictfile)
a = marshal.load(aaa)
aa = marshal.load(aaa)
aaa.close()
#aaa = open(dictfile)
#a = cPickle.load(aaa)
#aa = cPickle.load(aaa)
#aaa.close()
#a = shelve.open(rankfile, 'r')
#aa = shelve.open(dictfile, 'r')
#print 'phase', 1

arg1 = sys.argv[-2]  # first parameter
arg2 = sys.argv[-1]  # second parameter
```

```
def getwds(arg, b = {}, b_wds = []):
  numin = os.popen(datapat % (arg,))
  numfile = numin.read()      # read input file
  numin.close()
  b_lines = string.split(numfile, sep)[:-1]
  for line in b_lines:
    line = string.lower(line)  # apply transformations
    line = rulre1.sub(rep1, line)
    line = rulre2.sub(rep2, line)
    line = rulre3.sub(rep3, line)
    line = rulre4.sub(rep4, line)
    line = rulre5.sub(rep5, line)
    line = string.capwords(line)
    wds = string.split(line)  # split data into words
    prs = []      # word pairs
    tri = []      # word triples
    wdln = len(wds)
    i = 0
    while i < wdln:      # search word occurences
      if b.has_key(wds[i]):
        b[wds[i]] = b[wds[i]] + 1
      else:
        b[wds[i]] = 1
      i = i + 1
    i = 1
    while i < wdln:      # search word pair occurences
      i = i + 1
      pr = string.join(wds[i-2:i])
      if b.has_key(pr):
        b[pr] = b[pr] + 1
      else:
        b[pr] = 1
      prs.append(pr)
    i = 2
    while i < wdln:      # search word triple occurences
      i = i + 1
      tr = string.join(wds[i-3:i])
```

```
      if b.has_key(tr):
        b[tr] = b[tr] + 1
      else:
        b[tr] = 1
      tri.append(tr)
    b_wds.append((wds, prs, tri))
  return (b, b_wds)      # returns all relevant terms


def simwds(bx, u = []):    # compute most relevant phrase terms
  b, b_wds = bx
  nl = len(b_wds)
  i = 0
  while i < nl:
    nw = len(b_wds[i][0])
    j = 0
    q = []
    while j < nw:
      t = b_wds[i][0][j]
      tt = t
      if not a.has_key(t) and aa.has_key(t):
        tt = aa[t]
      if a.has_key(tt):
        p = -a[tt][0] / ((math.log(j + 1) + 1) * b[t])
        q.append((p, t))
      if j < nw - 1:
        t = b_wds[i][1][j]
        tt = t
        if not a.has_key(t) and aa.has_key(t):
          tt = aa[t]
        if a.has_key(tt):
          p = -a[tt][0] / ((math.log(j + 1) + 1) * b[t])
          q.append((p, t))
      if j < nw - 2:
        t = b_wds[i][2][j]
        tt = t
        if not a.has_key(t) and aa.has_key(t):
```

```
          tt = aa[t]
        if a.has_key(tt):
          p = -a[tt][0] / ((math.log(j + 1) + 1) * b[t])
          q.append((p, t))
      j = j + 1
    q.sort()    # sorts in order of importance
    u.append(q)
    #print b_lines[i], q[:3]
    i = i + 1
  return u


def chkwds(u, b = {}, z = []):    # eliminate duplication of terms
  for items in u:
    for item in items[:3]:
      t = item[1]
      if not b.has_key(t):
        b[t] = 1
        z.append(t)
  return z


def dowds(arg):
  return chkwds(simwds(getwds(arg)))

x = dowds(arg1)        # relevant terms from source 1
#print 'phase', 2
y = dowds(arg2)        # relevant terms from source 2
#print 'phase', 3

d = []          # comparison phase
for xtem in x:
  e = []
  for ytem in y:      # all term pairs from the two sources
    z = 0
    if xtem != ytem:
      z = 1
```

```
      q = 1
      for item in a[aa[xtem]][1:]:  # compute similarity
        tt = -1
        if item == ytem:
          z = z - (15.0 / q)
        elif item == '':
          q = 1
        else:
          q = q + 1
          if item in a[aa[ytem]][1:]:
            t = a[aa[ytem]][1:].index(item)
            if a[aa[ytem]].count('') == 2:
              tt = a[aa[ytem]][2:].index('')
              if t > tt:
                t = t - tt
            tt = (14.0 - abs(q - t)) / q
            z = z - tt
    e.append((z, ytem))
  e.sort()      # sort by strength of similarity
  d.append(e)      # prepare for output phase

#a.close()
#aa.close()
#print 'phase', 4


print comppat % (arg1, arg2)    # output phase
i = -1
for xtem in x:
  j = 0
  i = i + 1
  for item in d[i]:
    j = j + 1
    if item[0] < 0:    # only print if non-null similarity
      print rankpat % (xtem, j, item[1])
```

Appendix C

Semantic Heterogeneity in Literature and Art

Finally, we dedicate a section to the peripheral work that illustrates the issues of semantic heterogeneity more succinctly and clearly than any examples or toy problems can. An insightful analysis of Magritte's paintings, written by the philosopher Foucault, appears in [Fou73]. The worlds displayed in Magritte's work, called *heterotopias* by Foucault, are powerful examples of semantic mismatch. In particular, Magritte's use of lexical components within the art is disconcerting, because we are irresistibly drawn to interpret it. Doing so immediately changes the meaning of the entire piece within which the words appear.

Figure C.1: Upside Down

Another strongly visual example of the semantic issues resulting from the inclusion of text into art is a design by Scott Kim in Inversions [Kim81]. The design, shown in Figure C.1, when taken as text reads 'upside down,' no matter which orientation you give it. Other works that contain multiple heterogeneous meanings within single lexical domains are listed with brief explanations below.

Lewis Carroll's Jabberwocky [Car90]

> 'Twas brillig, and the slithy toves
> Did gyre and gimble in the wabe;
> All mimsy were the borogoves,
> And the mome raths outgrabe.

The verses of Jabberwocky are infinitely interpretable, since most of the terms are invented. The invented terms are, however, so strongly reminiscent of actual English words that we do attempt to make sense of them. It is remarkable that the verb 'to chortle' entered into the English language as a result of appearing in this poem.

Howard Chace's Ladle Rat Rotten Hut [Cha56]

> Wants pawn term, dare worsted ladle gull hoe lift wetter murder
> inner ladle cordage, honor itch offer lodge, dock, florist.
> Disk ladle gull orphan worry putty ladle rat cluck wetter ladle rat hut,
> an fur disk raisin pimple colder Ladle Rat Rotten Hut.

This fairy tale, reinterpreted, makes no sense at all when read literally. Every word in the text is lexically English, but there is no accompanying grammar to make meaningful sentences out of the words. However, when read out loud, or even better, having the text read out loud by someone else, makes the words sound like the original tale:

> Once upon a time, there was a little girl who lived with her mother...

C. van Rooten's Mots d'Heures, Gousses, Rames [van67]

Raseuse arrête, valet de Tsar bat loups	Roses are red, violets are blue
Joues gare et suite, un sot voyou	Sugar is sweet and so are you

The above verse, when read aloud with proper French pronunciation, sounds quite like the well known Mother Goose nursery rhyme shown to its right. The French text is puzzling, somewhat fanciful or archaic, yet it is grammatical. It is astonishing that it could even be possible for such completely different dual meanings to coexist in a single source of information. However, this example just amplifies what is common in every typical information source; different interpretations of the data results in different extracted meaning.

Smullyan's logic puzzles [Smu87]

> Two cards have *successive, positive integers* written on them.
> Two logicians each take one of them, without seeing the other's card.
> The ensuing dialog, in which each logician speaks in turn, is a sequence of n statements,
> A: I don't know your number
> B: I don't know your number
> ...
> A or B: I know your number
> What are the numbers on the cards?

This logical puzzle is an example of how additional information collapses two distinct world views into a coherent state. Simply stating, back and forth, their ignorance of the cards values, the logicians eventually rule out one of the possible values of their colleague's card. If A makes the n^{th} statement, the cards show the values $[n, n+1]$; if B makes the last statement the cards have the values $[n, n-1]$.

C.1 Limitations in Management of Semantic Heterogeneity

These examples all bring us back to the mirages, optical illusions mentioned the beginning of the introduction to the dissertation. While they appear to be extreme cases, they are in fact just indicative of the range of semantic heterogeneity that is in fact present in ordinary information sources. We have been trained to work only with data sources that are regular and well defined, and conditioned to think that every new source we encounter will follow the same pattern. Much of the work in this dissertation has reinforced a realization of how even the best maintained information sources are limited in their accuracy. The limitation seems to be simply the extent to which the information is used in practice. Every real world information source examined for this dissertation contained substantial inaccuracies, before even considering whether the content was appropriate for the tasks to which we applied it. The SKEIN system was designed to be built and to be used within this framework of uncertainty. The lesson learned is that no work of this magnitude is ever complete, and indeed, this dissertation and SKEIN are no exceptions. The best we can hope for is to be able to reach an acceptable level of refinement.

Bibliography

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semi-Structured Data*, pages 83–90. NSF, 1997.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[AM97] P. Atzeni and G. Mecca. Cut and paste. In *Proceedings, 17th ACM Symposium on Principles of Database Systems*, pages 144–153, June 1997.

[Ams80] R. Amsler. *The Structure of the Mirriam Webster Pocket Dictionary.* PhD thesis, University of Texas, Austin, 1980.

[ATT99] ATT Research. Graphviz. http://www.research.att.com/sw/tools/graphviz/, 1999.

[BCV99] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM SIGMOD '96*, pages 505–516. ACM, 1996.

[BHA96] C. J. Breiteneder, M. Hitz, and Mueck T. A. Metadata mining in legacy data sets. In *First IEEE Metadata Conference*, pages 48–57. IEEE, April 1996.

[BJBB+97] R. J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Infosleuth: Agent-Based semantic integration of information in open and dynamic environments. In *ACM SIGMOD '97*, pages 195–206. ACM, 1997.

[BLP00] P. A. Bernstein, A. Y. Levy, and R. A. Pottinger. A vision for management of complex models. Technical report, Microsoft Research, June 2000.

[Bri97] British Airways. British airways – flights. http://www.us.british-airways.com/flights/, July 1997. HTML table of BA Flights from San Francisco to London.

[Bri98] S. Brin. Extracting patterns and relations from the world wide web. In *Proceedings of International Workshop on the Web and Databases WEBDB*, 1998. http://www-db.stanford.edu/~sergey/booklist.html.

[Car90] L. Carroll. *Alice's adventures in wonderland / More annotated Alice.* Random House, New York, NY, 1990.

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *ACM SIGMOD '98*, pages 177–188. ACM, 1998.

[Cen97] Central Intelligence Agency. CIA site. http://www.odci.gov/cia, November 1997. CIA Factbook 1996.

[CF99] W. Cohen and W. Fan. Learning page independent heuristics for extracting data from web pages. In *Proceedings, 8^{th} International World Wide Web Conference WWW8*, May 1999.

[CG97] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *ACM SIGMOD '97*, pages 26–37. ACM, 1997.

[Cha56] H. L. Chace. *Anguish Languish.* Prentice Hall, Englewood Cliffs, NJ, 1956.

[CHR97] H. Chalupsky, E. Hovy, and T. Russ. NCITS.TC.T2 ANSI ad hoc group on ontology. Talk on Ontology Alignment, November 1997.

[CJN+00] A. Crespo, J. Jannink, E. Neuhold, M. Rys, and R. Studer. A survey of semi-automatic extraction and transformation. Technical report, Stanford University, 2000.

[Coh98] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD '98*, pages 201–212. ACM, 1998.

[COS98] K. E. Campbell, D. E. Oliver, and E. H. Shortliffe. The unified medical language system: toward a collaborative approach for solving terminologic problems. *Journal of the American Medical Informatics Association*, 5(1):12–16, 1998.

[Dai86] D. P. Dailey. On the search for semantic primitives. *Computational Linguistics*, 12(4):306–307, 1986.

[DDL+90] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. http://www.research.att.com/~mff/xml/w3c-note.html, August 1998. Working Draft.

[DH99] J. Dean and M. R. Henzinger. Finding related pages in the world wide web. In *Proceedings, 8^{th} International World Wide Web Conference WWW8*, May 1999.

[DHR+98] R. H. Dolin, S. M. Huff, R. A. Rocha, K. A. Spackman, and K. E. Campbell. Evaluation of a "lexically assign, logically refine" strategy for semi-automated integration of overlapping terminologies. *Journal of the American Medical Informatics Association*, 5(2):203–213, 1998.

[DJ00] S. Decker and J. Jannink. Toward formal ontology algebras: First steps. Technical report, Stanford University, 2000.

[DMW00] S. Decker, S. Melnik, and G. Wiederhold. Ontoagents project. http://www-db.stanford.edu/OntoAgents/, 2000.

[Enc99] Encyclopedia Britannica. Encyclopedia britannica online. http://www.eb.com/, 1999.

[FFLS98] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Reasoning about web-site structure. In *AAAI Workshop on AI and Information Integration*, pages 505–516. AAAI, July 1998.

[Fou73] M. Foucault. *Ceci n'est pas une pipe.* Editions fata morgana, Montpellier, France, 1973. trans: This is not a Pipe, Harkness, J., U. C. Press, 1983.

[Fow98] M. Fowler. *UML Distilled.* Addison–Wesley, Reading, MA, 1998.

[GCCM96] R. Goldman, S. Chawathe, A. Crespo, and J. McHugh. A standard textual interchange format for the object exchange model. http://www-db.stanford.edu/pub/papers/oemsyntax.ps, 1996.

[Ger96] A. Geraci. The computer dictionary project: An update. *Computer*, 29(7):95, July 1996.

[Gig00] Gigabeat. Discover new music. http://www.gigabeat.com/disc/, June 2000. Visualization of Relationships Between Artists and Songs.

[GKD97] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster: An information integration system. In *ACM SIGMOD '97*, pages 539–542. ACM, 1997.

[GN88] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Palo Alto CA, 1988.

[Goo99] Google. Google search. http://www.google.com/, 1999.

[GSVG98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity searching in databases. In *Proceedings of the 24th VLDB Conference*, pages 26–37. VLDB, 1998.

[Guh91] R. V. Guha. *Contexts: A Formalization and Some Applications.* PhD thesis, Stanford University, 1991.

[GW99] T. Guan and K.-F. Wong. KPS: a web information mining algorithm. In *Proceedings, 8^{th} International World Wide Web Conference WWW8*, May 1999.

[HM93] J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(1):51–83, March 1993.

[HP98] M. C. Horsch and D. Poole. An anytime algorithm for decision making under uncertainty. In *14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 246–255. AUAI, July 1998.

[Hul97] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. http://www-db.research.bel-labs.com/user/hull/pods97-tutorial.html, 1997. tutorial slides & references.

[Jan99a] J. Jannink. Thesaurus entry extraction from an on-line dictionary. In *Proceedings, Second International Conference on Information Fusion.* ISIF, 1999. http://www-db.stanford.edu/SKC/papers/thesau.ps.

[Jan99b] J. Jannink. Webster's dictionary nexus. http://skeptic.stanford.edu/data/, 1999.

[Jan00] J. Jannink. Oxford english dictionary nexus. http://skeptic.stanford.edu/data/oed.html, 2000.

[JMN+99] J. Jannink, P. Mitra, E. Neuhold, S. Pichai, R. Studer, and G. Wiederhold. An algebra for semantic interoperation of semistructured data (extended version). In *IEEE Knowledge and Data Engineering Exchange Workshop, KDEX '99.* IEEE, November 1999.

[JPVW98] J. Jannink, S. Pichai, D. Verheijen, and G. Wiederhold. Encapsulation and composition of ontologies. In *AAAI Workshop on AI and Information Integration.* AAAI, 1998. http://www-db.stanford.edu/SKC/publications/jpvw_encaps.ps.

[JW99] J. Jannink and G. Wiederhold. Ontology maintenance with an algebraic methodology: a case study. In *Proceedings, AAAI Workshop on Ontology Management.* AAAI, 1999. http://www-db.stanford.edu/SKC/papers/summar.ps.

[Kar92] P. D. Karp. The design space of frame knowledge representation systems. Technical report, SRI International Artificial Intelligence Center, 1992.

[Kim81] S. Kim. *Inversions.* BYTE Books, Peterborough, NH, 1981. see also: http://www.scottkim.com/.

[Kle98] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998. http://simon.cs.cornell.edu/home/kleinber/auth.ps.

[Knu97] D. E. Knuth. *Stable Marriage and its Relation to Other Combinatorial Problems.* American Mathematical Society, Providence RI, 1997.

[KS96] V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: a context-based approach. *VLDB Journal*, 5(4):276–304, October 1996.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference*, pages 251–262. VLDB, 1996.

[LW01] K. Law and G. Wiederhold. Regnet and regbase regulation interoperation projects. http://eil.stanford.edu/law/home/research.htm, 2001.

[MB95] G. Mecca and A. J. Bonner. Sequences, datalog and transducers. In *Proceedings, 15th ACM Symposium on Principles of Database Systems*, pages 23–35, May 1995.

[MBF+90] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five papers on WordNet. Technical Report 43, Cognitive Science Laboratory, Princeton University, 1990.

[MIC96] MICRA inc. Webster's dictionary, 1913. ftp://ftp.uga.edu/pub/misc/webster/, 1996. also available from Project Gutenberg.

[Mir99] Mirriam-Webster. WWWebster dictionary. http://www.m-w.com/, 1999.

[MKW00] P. Mitra, M. Kersten, and G. Wiederhold. A graph-oriented model for articulation of ontology interdependencies. In *Proceedings, 7th International Conference on Extending Database Technology*, pages xx–xx, March 2000.

[MMK98] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured, web-based information sources. In *Proceedings of the Conference on Automatic Learning and Discovery, CONALD-98*, 1998.

[MP95] R. Motwani and Raghavan P. *Randomized algorithms.* Cambridge University Press, New York NY, 1995.

[myS99] mySimon. mysimon inc. - mysimon comparison shopping. http://www.mysimon.com/, 1999.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *ACM SIGMOD '98*. ACM, June 1998.

[Nas51] J. F. Nash. Non-cooperative games. *Annals of Mathematics*, 54:286–295, 1951.

[NAT99] NATO. Partnership for Peace. http://www.nato.int/pfp/partners.htm, 1999.

[Net99a] Netscape Inc. Netscape netcenter. http://www.netscape.com/, 1999.

[Net99b] Netscape Inc., Open Directory Team. Open Directory Project. http://www.dmoz.org/, 1999.

[NTR98] R. Nikolai, A. Traupe, and Kramer R. Thesaurus federations: A framework for the flexible integration of heterogeneous, autonomous thesauri. In *Proc. Conference on Research and Technology Advances in Digital Libraries (ADL'98)*, pages 46–55, 1998.

[OSSM99] D. E. Oliver, Y. Shahar, E. H. Shortliffe, and M. Musen. Representation of change in controlled medical terminologies. *Artificial Intelligence in Medicine*, 15(1):53–76, January 1999.

[Oxf99] Oxford University Press. Oxford english dictionary. http://www.oed.com/, 1999.

[PB98] L. Page and S. Brin. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the 7th Annual World Wide Web Conference*, 1998.

[PG95] Y. Papakonstantinou and H. Garcia-Molina. Object fusion in mediator systems (extended version). Technical report, Stanford University, 1995.

[PHW+99] P. Panchapagesan, J. Hui, G. Wiederhold, S. Erickson, L. Dean, and A. Hempstead. The INEEL data integration mediation system. In *Proceedings, International ICSC Symposium on Advances in Intelligent Data Analysis, AIDA '99*, 1999.

[Plu00] Plumb Design. WordNet applet. http://www.plumbdesign.com/thesaurus/, 2000. Applet to Visualize the Relationship Between WordNet Terms.

[Pra97] W. Pratt. Dynamic organization of search results using the UMLS. In *Proceedings, Amia Annual Fall Symposium*, pages 480–484, 1997.

[pro99a] CLEVER project. CLEVER searching. http://www.almaden.ibm.com/cs/k53/clever.html, August 1999. HITS, Hubs and Authorities research.

[PRO99b] PROMO.NET. Project Gutenberg. http://www.gutenberg.net/, 1999.

[RBK+00] S. Rajagopalan, A. Broder, R. Kumar, F. Maghoul, P. Raghavan, R. Stata, A. Tomkins, and J. Weiner. The web as a graph: structure and interpretation. http://www.almaden.ibm.com/cs/k53/clever.html, March 2000. presentation at Stanford University.

[RDV98] S. Richardson, W. Dolan, and L. Vanderwende. MindNet: acquiring and structuring semantic information from text. In *Proceedings of COLING '98*, 1998. ftp://ftp.research.microsoft.com/pub/tr/tr-98-23.doc.

[Res98] CL Research. Dictionary parsing project. http://www.clres.com/dpp.html, 1998.

[Ric97] S. D. Richardson. *Inferring Relations in a Lexical Knowledge Base*. PhD thesis, City University of New York, 1997.

[RS91] J. Richardson and P. Schwarz. MDM: An object-oriented data model. In *Proceedings, Third International Workshop on Database Programming Languages*, pages 86–95, 1991.

[SM91] M. Siegel and S. E. Madnick. A metadata approach to resolving semantic conflicts. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 133–145, 1991.

[Smi93] D. R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15:571–606, 1993.

[Smu87] R. M. Smullyan. *Forever undecided : a puzzle guide to Gödel*. Knopf, New York, NY, 1987.

[Sow00] J. F. Sowa. *Knowledge Representation Logical, Philosophical and Computational Foundations*. Brooks/Cole, Pacific Grove, CA, 2000.

[Tom99] F. Tompa. Centre for the New OED and Text Research. http://db.uwaterloo.ca/OED/, 1999.

[UHW+98] M. Uschold, M. Healy, K. Williamson, P. Clark, and S. Woods. Ontology reuse and application. In *Formal Ontology in Information Systems, FOIS'98*, 1998.

[Uni97] United Airlines. United airlines flight search. http://flights.ual.com/cgi-bin/ua_search/flight_grep.cgi/style=hi, July 1997. HTML table of UA Flights from San Francisco to London.

[van67] C. H. K. van Rooten. *Mots d'heures, gousses, rames.* Penguin Books, New York, NY, 1967.

[VP99] V. Vasalos and Y. Papakonstantinou. Query rewriting for semistructured data. In *ACM SIGMOD '99.* ACM, 1999.

[Wie94] G. Wiederhold. An algebra for ontology composition. In *Proceedings of 1994 Monterey Workshop on Formal Methods*, pages 56–61. U.S. Naval Postgraduate School, September 1994.

[WM00] P. Walms and J. P. Morgenthal. Mining for metadata. *Software Magazine*, 2(2):12–18, 2000.

[ZE99] O. Zamir and O. Etzioni. Grouper: a dynamic clustering interface to web search results. In *Proceedings, 8^{th} International World Wide Web Conference WWW8*, May 1999.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison–Wesley, Cambridge, MA, 1949.