Applying Semantic Relation Extraction to Information Retrieval

by

Edward Loper

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2000

Certified by .....
Boris Katz
Principal Research Scientist
Thesis Supervisor

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

Applying Semantic Relation Extraction to Information Retrieval

by

Edward Loper

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

People often want to search a large number of articles, web pages, or other natural-language documents to find the answers to specific questions. Currently, the best techniques for performing such searches treat each document as an unordered collection of words, and disregard any information about how the words relate to each other. Although this approach has been quite successful, it has a number of fundamental limitations. In order to explore the potential benefit of using more advanced, linguistically-motivated techniques, I built SQUIRE, a Semantic Question-answering Information Retrieval Engine. SQUIRE uses a number of independent transformations to process individual sentences, converting them into sets of semantic relationships. It can then search those semantic relationships in response to a user's question. SQUIRE significantly out-performs traditional IR techniques in the task of returning the individual sentences that answer a question.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist

Acknowledgments

I would like to thank Boris Katz and Sue Felshin, who helped to shape my thesis topic. I would also like to thank everyone who read and commented on early drafts of my thesis: Jee, Emil, Sue, and Boris.

Contents

1 Introduction **10**

1.1 Searching Natural Language Databases 10

1.1.1 Traditional Information Retrieval Techniques 10

1.1.2 Limitations of Traditional Information Retrieval 11

1.1.3 Semantic Information Retrieval 12

1.2 Overview 14

2 Background Information **16**

2.1 Information Retrieval Techniques 16

2.1.1 Removing Functional Words 16

2.1.2 Stemming 17

2.1.3 Weighting 19

2.1.4 Vector Space Model 21

2.2 Measuring Information Retrieval Performance 21

2.3 Linguistics 22

2.3.1 Syntax Trees 23

2.3.2 Arguments and Modifiers 27

2.3.3 Thematic Roles 28

2.3.4 Alternations 29

2.3.5 Verb Classes 30

2.3.6 Predicate Logic 31

2.3.7 Fregean Semantics 32

2.3.8 Semantic Trees 33

2.4 START 33
2.4.1 The Structure of Links 35

3 Exploring Semantic IR **37**
3.1 SQUIRE's Task 37
3.2 Design Criteria 38
3.2.1 Response Criteria 38
3.2.2 System Properties 40

4 SQUIRE: A Semantic Information Retrieval Engine **41**
4.1 Weights 41
4.2 The Parser 42
4.2.1 Classifications of Parser Layers 43
4.2.2 Advantages of the Layered Architecture 44
4.2.3 Parsing in Parallel 46
4.3 Representations 46
4.3.1 Strings 47
4.3.2 Syntax Trees 47
4.3.3 Semantic Trees 48
4.3.4 Relations 49
4.4 Parser Layers 50
4.4.1 The Principar Layer 50
4.4.2 The Stemmer 52
4.4.3 The Clause Handler 52
4.4.4 The Conjunction Handler 53
4.4.5 The Trace Resolver 55
4.4.6 The Semantic Tree Maker 55
4.4.7 The Pronoun Resolver 59
4.4.8 The Property Adder 60
4.4.9 The Theta Assigner 61
4.4.10 The Synonymy Layer 63

4.4.11 The Relation Maker 65
4.5 The Database 65
4.5.1 Indexing 66
4.5.2 Retrieval 68

5 Testing **70**
5.1 Test Materials 70
5.1.1 Test Corpus 70
5.1.2 Test Questions 70
5.1.3 Test Indexes 71
5.2 Methodology 71
5.3 Control 73
5.4 Results 73
5.4.1 Effects of the *word* parameter 73
5.4.2 Effects of the *ur* parameter 75
5.4.3 Effects of the *fr* parameter 76
5.4.4 Effects of the *pro* parameter 78
5.4.5 Effects of the *syn* parameter 80
5.5 Conclusions 81

6 Future Directions **83**
6.1 Improving Existing Parser Layers 83
6.2 Creating New Parser Layers 84
6.3 Fundamental Changes to SQUIRE 85

List of Figures

1-1 Sentence Retrieval: Indexing Sentences as Simple Structures 15

2-1 Example Syntax Tree 23
2-2 Semantic Phrases with Lexical Heads 24
2-3 Semantic Phrases with Functional Heads 24
2-4 Illustration of Syntactic Traces 26
2-5 Illustration of *PRO* 27
2-6 Commonly Used Thematic Roles 29
2-7 Example of Fregean Semantic Derivation 32
2-8 Example Semantic Tree 34

4-1 Example Parser 43
4-2 Indexing the Relations in a Sentence 49
4-3 Indexing Words as Relations 50
4-4 SQUIRE's Parser Layers 51
4-5 Example Templates for the Semantic Tree Maker 56
4-6 Example Template for the Theta Assigner 62
4-7 Database Structure 66
4-8 Database Indexing 67
4-9 Database Retrieval 69

5-1 Precision-Recall Curve of the Control IR System 72
5-2 Comparison of Control IR System to SQUIRE Indexing Words 74

5-3 Precision-Recall Curve of SQUIRE Using the Pronoun-Resolution and Synonymy Layers to Index Words 74
5-4 Precision-Recall Curve of SQUIRE Indexing Underspecified Relations. 75
5-5 Precision-Recall Curve of SQUIRE Indexing Both Words and Underspecified Relations. 76
5-6 Comparison of Indexing Underspecified Relations to Indexing Tagged Relations 77
5-7 Precision-Recall Curves Illustrating the Effect of the Pronoun-resolution Parser Layer 79
5-8 Precision-Recall Curves Illustrating the Effect of the Synonymy Layer when Relations are Indexed 80
5-9 Precision-Recall Curves Illustrating the Effect of the Synonymy Layer when Only Words are Indexed 81
5-10 Comparison of the Performance of SQUIRE with All Features Enabled to the Performance of the Control IR System 82

Chapter 1

Introduction

1.1 Searching Natural Language Databases

People often want to search a large natural-language reference source, such as an encyclopedia, to find the answers to specific questions. However, finding these answers can be a daunting task. Even when it is relatively certain that the answer is contained somewhere in the reference, it may not be apparent *where* the answer is stored. It is especially difficult to find answers in references that are large, that contain a wide variety of information, and that do not have well-defined organizations.

An example of a reference source with all of these properties is the World Wide Web. In general, it is very difficult to extract specific information from the web, since it is such a large, varied, and unorganized source of data. Other examples include news wires, scientific journals, and encyclopedias.

1.1.1 Traditional Information Retrieval Techniques

The task of automatically searching a large reference for answers to a specific question is known as **information retrieval** (IR). Information retrieval systems can be classified by what types of answers they return, and by how they process queries to arrive at those answers.

Traditional IR systems divide reference sources into sets of **articles**, or small (100-

5,000 word) self-contained documents, each of which discusses a specific topic. Given a question, they will return the set of articles which they find most likely to contain the answer to that question. To decide which articles answer a question, they model the question and all the articles in the database as unordered "bags of words." They then return the articles whose "bags of words" are most similar to the question's. This approach is exemplified by the SMART IR system [23].

1.1.2 Limitations of Traditional Information Retrieval

Although the traditional IR approach has been very successful, it has a number of important limitations.

1. By treating a document as an unordered set of words, the traditional IR approach throws away all information about how the words relate to each other. As a result, it will return documents that don't answer the question but that have similar word content. A few examples of phrases which have different meanings but similar word content are:

The man ate the dog	The dog ate the man
The meaning of life	A meaningful life
The bank of the river	The bank near the river

2. The set of words used to answer a question are not necessarily the same as those used to ask it. As a result, the traditional approach will fail to return documents that answer the question using different words than the question itself. For example, the following pairs of questions and answers use very different words:

   - What color is the sky?
     When viewed from the surface of the earth, the atmosphere looks pale blue.
   - Which European discovered America?
     Columbus found the New World in 1492.
   - What do pigs eat?
     Hogs feed on corn.

3. Individual sentences contain only a few words, and use pronouns and other devices to refer to previously mentioned concepts. In fact, the word content of a question is very rarely similar to the word content of the sentence that answers it. This exacerbates the first two problems, and means that traditional IR techniques are very poor at deciding which sentence or paragraph from an article answers a question. For example, a traditional IR system would have no hope of discovering the following answers for questions, since they make only implicit reference to the objects in question:

   - Who invented basketball?
     It was initially conceived of by James Naismith.
   - How many people live in Washington, D.C.?
     The city has a population of 600,000 people.
   - What is grown in the Dominican Republic?
     The fertile ground is farmed to produce oranges and rice.

These limitations are all fundamental consequences of the fact that traditional IR techniques treat queries and documents as unordered sets of words. In order to overcome them, information about the structure of the documents and queries must be extracted and stored.

1.1.3 Semantic Information Retrieval

Intuitively, we would like to be able to index all of the "knowledge" from a reference source. Questions could then be turned into "knowledge templates," that could be matched against the indexed knowledge. However, before we can construct such an index, we must answer three related questions:

1. How should we represent knowledge and questions?

2. How can we extract that representation from a reference source or a question?

3. How can we index and efficiently search the knowledge we extract?

Representation

Since sentences are, in a sense, the basic unit of knowledge, each sentence should have its own representation. Each sentence could be represented with a single complex structure, or with a set of several simpler structures. Larger structures tend to capture more information about the original content of the sentence, while simpler structures are usually easier to index and search.

Most semantically based information retrieval systems represent articles and references as unordered sets of sentences [12, 2], and do not encode information about how those sentences relate to each other.

The knowledge representation should optimally satisfy the following properties:

- **Canonical Representation:** Two different sentences with the same basic meaning should have the same representation.
- **Generalization:** If one sentence implies another, then the knowledge representation of the first should either be contained in the second, or be derivable by the retrieval algorithm.

Assuming that the knowledge representation satisfies these properties, we can simply represent questions as partial sentences, or as sentences with variables in them. Note, however, that if the knowledge representation is a complex, nested structure, then deciding which sentences answer a question may require individually processing each sentence in the database.

Parsing

Parsing is the process of constructing the representation that corresponds to a sentence. There are two basic approaches to parsing a sentence:

- In **Deep Parsing**, the sentence is first converted into a linguistically motivated tree-shaped structure. This structure explicitly encodes how different parts of the sentence relate to each other, and removes most ambiguity from the original

sentence. This structure is then used to construct a knowledge representation of the sentence.

- In **Surface Parsing**, the sentence is treated as a sequence of words, and the knowledge representation is constructed directly from this sequence. Usually, this construction involves matching sequences of parts of speech against the sentence [12]. For example, if a surface parser encounters an adjective followed by a noun, it will encode the adjective as a modifier of the noun.

Indexing and Retrieval

The representations of a reference source's sentences must be indexed in such a way that they can be efficiently searched. In particular, searching the index should not require that a question be compared to every sentence. This can be a problem for complex knowledge representations, since it is hard to index them such that they will be returned for just those questions that they answer.

For that reason sentences are usually broken up into sets of simple structures, which are indexed separately. To find the answers to a question, the index is searched for structures that match the question's structures. The source sentences of these matching structures are then returned as possible answers to the question. Source sentences that contain more matching structures are rated as more likely to be correct answers. This process is illustrated in Figure 1-1.

1.2 Overview

Although semantic information retrieval techniques have always seemed very promising, traditional techniques have continually out-performed them [28, 2]. In order to explore the limitations and the potential of semantic IR techniques, I designed and built SQUIRE, a Semantic Question-Answering IR Engine. SQUIRE uses deep parsing techniques to convert sentences into sets of simple structures, representing the individual semantic relationships that comprise them. These relationships are then separately indexed in a database. When the user asks SQUIRE a question, SQUIRE

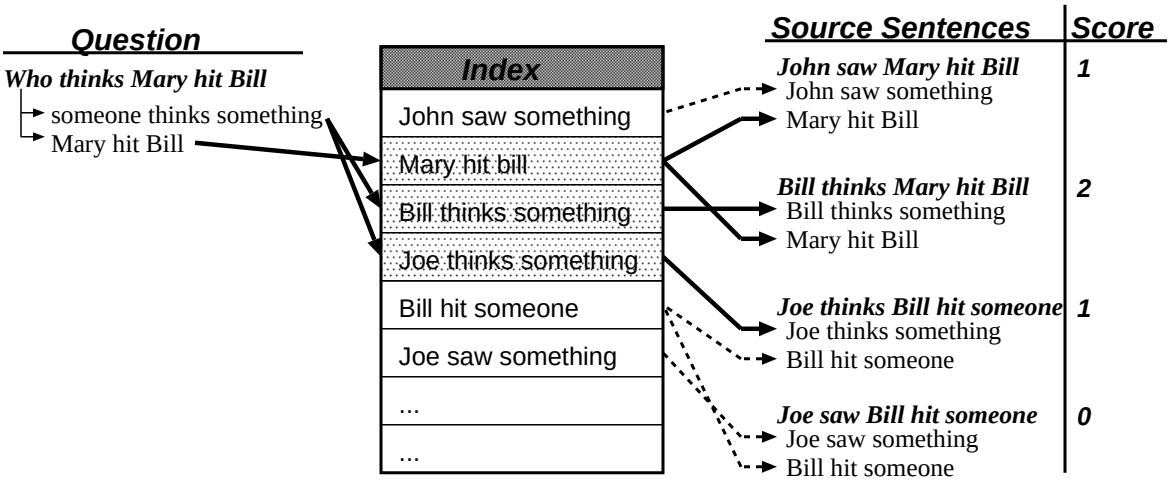

Figure 1-1: An example of sentence retrieval in a system where sentences are indexed as simple structures. In this case, the sentence is reduced to subject-verb-object triples. The question is first broken into two relations. Those relations are then looked up in the index, and the sentences which contain them are returned.

converts it into a set of semantic relationships, and searches the database for the sentences which best match those relationships.

This thesis describes the design and implementation of SQUIRE, and examines its performance using a test set of thirty-eight questions. Chapter 2 provides some background information about traditional information retrieval techniques and about linguistics. Chapter 3 describes the specific problem that SQUIRE tries to solve, and the properties that it should have to solve that problem. Chapter 4 describes SQUIRE's design and implementation. Chapter 5 describes the tests I ran to compare SQUIRE to traditional IR techniques, and discusses the results. Chapter 6 explores some future directions that the research into Semantic IR might take.

Chapter 2

Background Information

2.1 Information Retrieval Techniques

Traditional information retrieval systems treat both articles and questions as unordered sets of words. When given a question they return the articles whose representative sets of words best match the question's. Although this general description is true of most traditional IR systems, it leaves a number of details unspecified:

- What words are put in the unordered sets used to represent articles?
- How does the system decide how well two sets of words "match?"
- Are weights associated with the words in the set? If so, what do they represent?

A number of answers have been given to each of these questions. In the following sections, I will describe the answers which seem to maximize performance in traditional information retrieval systems. Most of these techniques have become standard, and are incorporated into almost every traditional IR system.

2.1.1 Removing Functional Words

Some words give us more information about the semantic content of a sentence than others. This is important to traditional IR systems, since a word that does not

convey much semantic content will not be as likely to correlate between questions and answers. Words can be divided roughly into two classes, based on how well their presence in a question correlates with their presence in an article that answers the question:

- **Content words**, such as "fern," "carpet," and "elect" correlate well; if a question's content word is in an article, then we are more likely to believe that the article answers the question.
- **Functional words**, such as "of," "him," "the," and "throughout," do not correlate well; the presence of a question's functional word is in an article gives us no information about whether the article answers the question.

Since functional words do not provide traditional information retrieval techniques with any information, they are not usually included in the sets of words used to represent articles and questions; thus, the sets of words that represent articles contain only content words.

This is usually accomplished by using a **stop list**, which lists all of the functional words [27]. Any word in the stop list will not be included in the representation of the article or question. Note that it is unclear which words should be placed in a stop list, since some words have intermediate correlation, and many words have correlations that depend on the domains of the questions and of the articles. For example, when asking a question about Australia, it would probably be inappropriate to eliminate the words "down under," even though they appear in most stop lists.

2.1.2 Stemming

In English, many words in a sentence are given an affix, such as "-s" or "-es" for nouns, and "-ing" or "-ed" for verbs. However, these affixes are not part of the semantic content of the word itself. Instead, they give an indication of how the word relates to other words in the sentence.

It would be unreasonable for traditional IR systems to treat these affixes as part of the word. If it did, then only those articles whose words contain the same affixes

as the question would be returned. For example, it would not return "The cow eats grass" in response to "what do cows eat," since the affixes on both "cow" and "eat" do not match. For this reason, most IR systems use the root words, or **stems**, to encode articles and questions, and throw away any affixes. This process is known as **stemming**.

However, there are several different types of affixes in English, and it is not clear that all of them should be discarded. Affixes like "-s" in "car*s*" and "-ing" in "walk*ing*" serve to modify a basic concept. These are discarded by almost all stemmers. Affixes like "un-" in "*un*wanted," "-ize" in "national*ize*," and "-er" in "hunt*er*" can fundamentally change the meaning of their stem. Different stemmers make different decisions about which of these affixes should be kept and which can be discarded [27, 1].

It is not always clear how to decide what the stem of a word is, even once we decide what affixes we wish to remove. For example, a stemmer might want to strip the suffix "-tive" from "administra*tive*" to give "administrate," but it shouldn't also strip the same suffix from "additive" to give "addite" or from "adjective" to give "adject."

Most stemmers use a number of heuristics to produce the stems for a words. Typically, such stemmers have an error rate of about 5% [27]. Another simple approach is to simply maintain a very large list of every word that we are likely to encounter, and its stem. The disadvantage of this approach is that it will not stem words that it has not encountered, most likely including specialized words like "Minke" (a type of whale). It might be possible to combine these techniques – use a table of stems for most words, and only revert to the heuristic stemmer for words that are not present in the table.

Lemmatization

The proper stem for a word is often dependent on how it is used in a sentence. One technique for improving the performance of stemming is to first derive the part of speech of the word, and then to use that information to decide what stem the word

should be given. This technique is known as **lemmatization** [2]. In order to find the part of speech of the words in an article, a statistical algorithm is usually used. One popular algorithm for finding parts of speech of words is Brill's Tagger [5].

2.1.3 Weighting

In addition to storing the set of words that comprise an article, almost every information retrieval system also assigns a weight to each word [27, 23]. This weight generally reflects how much that word tells us about the content of the article. The IR system can then use those weights to decide how well a question matches an article.

Weighting by Frequency in the Article

By reducing an article to a set of words, information about how often a word occurs in an article is lost. In order to preserve this information, most IR systems use the weights of the words to reflect their frequency of in an article [27]. Representing this frequency as an absolute number of occurrences would unfairly give higher weights to larger articles. Therefore, this frequency is often represented as a percentage of total number of words in the article. Let f_{id} be the number of occurrences of word i in document d, and let n_d be the number of words in document d. Then w_{id}, the weight of word i in document d, is:

$$w_{id} = \frac{f_{id}}{n_d} \in [0, 1]$$

Weighting by Inverse Overall Frequency

In general, the more common a word is, the less likely it is to help decide whether an article is an answer to a question [27]. For example, a question about "heart defibrillation" is more likely to be answered by an article containing the word "defibrillation" than by one containing "heart," since "defibrillation" is a much less common word than "heart." For this reason, many IR systems take into account how common a word is to decide what weight it should receive. The commonness of a word can be

measured by its overall frequency in the corpus being indexed. Let N be the number of documents. Then the commonness of word i, $C_i \in [0, 1]$ is:

$$C_i = \frac{\sum_{j=1}^{N} f_{ij}}{N}$$

The commonness could also be measured using an independant corpus. Once the commonness of each word has been defined, we can find the weight of word i in document d:

$$w_{id} = \frac{f_{id}}{n_d \times C_i}$$

Other Weighting Schemes

A variety of weighting schemes are possible, which may take into account the frequency of a term, the number of documents it occurs in, and other information. One commonly used weighting scheme is the Cornell *ltc* weighting [8]:

$$w_{id} = \frac{(\log(f_{id}) + 1) \times \log(N/m_d)}{\sqrt{\sum_{j=1}^{N} (\log(f_{ij}) + 1) \times \log(N/m_j)}}$$

where m_d is the number of documents in which term d appears.

Weighting by Article Value

Sometimes, it is possible to determine through external means how reliable, informative, or useful a document is. In this case, we would like to return articles which score higher on these criteria. One way to accomplish this is to scale the weights of every term in an article's representation by an overall measure of the value of the article.

For example, the Google search engine[9] uses the heuristic that pages to which more people make links are more likely to be useful. They therefore modify the weights of the articles to reflect this information — articles which more people link to will be more likely to be returned, and articles that few people link to will be returned less often [6].

2.1.4 Vector Space Model

Instead of thinking of an article as a set of weighted word stems, we could think of it as a mapping from word stems to weights. The weight of any word stem not present in the article is simply zero. By defining a vector containing all of the word stems in the corpus:

$$WORD = \langle word_1, word_2, ..., word_i, ..., word_M \rangle$$

we can define the representation of an article or question to simply be the vector:

$$W_i = \langle w_{i1}, w_{i2}, ..., w_{iM} \rangle$$

where w_{id} is the weight of word i in document d.

This representation reduces each document to a single high-dimensionality vector. These vectors are a convenient representation, because they can be manipulated and reduced using standard linear algebra techniques. For example, since many words tend to co-vary in articles, we can find a minimal basis for the vector space (i.e., a minimal set of perpendicular axes that allow us to encode all of the vectors). This cuts down on the space the vectors take, and on the time it takes to compare them [27].

2.2 Measuring Information Retrieval Performance

Since the early 1960's, much research has gone into finding good metrics with which to measure information retrieval performance [25]. Since then, many alternative metrics have been proposed[25, 7, 22, 3]. However, only a few have become widely accepted. Most of the widely accepted metrics are based on the **contingency table**, which divides the sentences of the original corpus into four categories, based on whether they answer the question and on whether they are retrieved in response to the question [27]:

	answers question (A)	doesn't answer question ($\bar{A}$)
retrieved (B)	$A \cap B$	$\bar{A} \cap B$
not retrieved ($\bar{B}$)	$A \cap \bar{B}$	$\bar{A} \cap \bar{B}$

Given this table, we can define the two most commonly used metrics, precision and recall. **Precision** is the percentage of the responses retrieved that answer the question. **Recall** is the percentage of the correct answers that are retrieved:

$$precision = \frac{|A \cap B|}{|B|}$$

$$recall = \frac{|A \cap B|}{|A|}$$

In most information retrieval systems, there is a tradeoff between precision and recall. In particular, increasing the number of documents returned generally increases recall at the expense of precision, and decreasing the number returned increases precision at the expense of recall. At the very extremes, returning every document would give a recall of 1 and a precision of 0; and returning no documents would give a precision of 1 and a recall of 0.

Although there is a tradeoff between precision and recall, the nature of this tradeoff varies widely between different IR systems. Because the nature of the tradeoff varies so widely, it is not clear how to reduce these two quantities to a single value that indicates the "overall performance" of the system. Therefore, most researchers report the performance of an IR system using a graph of its precision as a function of its recall [27].

2.3 Linguistics

Although sentences may seem to simply consist of simple strings of words, linguists have long recognized that sentences have a much more complex underlying structure [11, 20]. This structure explicitly indicates how the constituent words of the sentence relate to each other, and how they combine to give the meaning of the sentence as a whole. In order to get an idea of what a sentence means, and whether

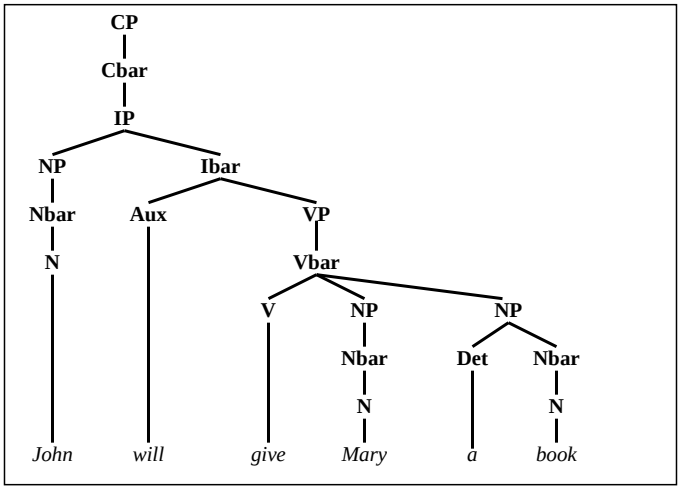

Figure 2-1: Example of a syntax tree, for the sentence "John will give Mary a book."

it answers a question, we will need to extract this underlying structure from the sequence of words that make up the sentence.

2.3.1 Syntax Trees

Most linguists agree that sentences are best represented by a tree-based structure, known as a **syntax tree** [11]. The leaves of a syntax tree are the words of the sentence; the nodes represent **phrases**, or groups of words and sub-phrases that combine to form a single functional unit. An example of a syntax tree is given in Figure 2-1.

Phrases

Each phrase consists of a **head**, which contains the main semantic idea of the sentence, and a number of **adjuncts**, which modify the head in some way. In addition, each phrase is assigned a type, based on what type of head it has. For example, phrases headed by nouns are **noun phrases** (NP), phrases headed by verbs are **verb phrases** (VP), and phrases headed by adverbs are **adverbial phrases** (AdvP). Some example phrases are given in Figure 2-2.

windows	(Noun Phrase)
the big **man** that I talked to yesterday	(Noun Phrase)
eat the cake	(Verb Phrase)
walk to the store to buy groceries	(Verb Phrase)
quickly	(Adverbial Phrase)

Figure 2-2: Each phrase consists of a head and zero or more arguments. The head is in bold face, and each argument is underlined. Each of these phrases has a lexical head (i.e., a head that represents an independent semantic concept).

to the moon	(Prepositional Phrase)
under the carpet	(Prepositional Phrase)
the man	(Determiner phrase)
John **will** walk the dog	(Inflection phrase)

Figure 2-3: Each phrase consists of a head and zero or more arguments. The head is written in bold face, and each argument is underlined. Each of these phrases has a functional head (i.e., a head that does not represent an independent semantic concept).

Heads

Heads can be divided into two classes: lexical heads and functional heads. **Lexical heads** represent an independent semantic concept. All of the phrases above have lexical heads [11]. **Functional heads** are heads which do not represent an independent semantic concept. Functional heads generally take exactly one adjunct, and serve to modify that adjunct [11]. Some examples of phrases with functional heads are given in Figure 2-3

Movement and Traces

Sometimes, the phrases that make up a sentence are moved from their original positions. For example, consider the following sentences:

(1) John likes Mary

(2) Mary is liked

whose syntax trees are shown in Figure 2-4, parts (a) and (b). In both sentences, "Mary" relates to the verb "to like" in the same way, although in sentence (1) "Mary" is an adjunct to "likes," while in sentence (2) "Mary" is an adjunct to "is."

Linguistic research suggests that, in sentence (2), "Mary" actually originated at the same position as it did in sentence (1), but was subsequently moved out of this position [11]. Thus, there are actually two syntax trees corresponding to sentence (2): a pre-movement tree and a post-movement tree. The post-movement tree is known as the **surface structure** of the sentence, since it reflects the ordering of words as they are pronounced. The pre-movement tree is known as the **deep structure** of the sentence, since it more closely reflects the semantic relations that make up the concept expressed by the sentence.

When a phrase is moved, its original location is marked with a special entity called a **trace**. Traces are not directly pronounced, but they can be detected by a number of means, including monitoring the timing of speech [26]. The displaced phrase is called the trace's **antecedent**. A connection is maintained between a trace and its antecedent, and the trace and antecedent are said to **corefer**. Parts (c) and (d) of Figure 2-4 illustrate the deep structure and the surface structure of sentence (2).

Non-overt Pronouns: *PRO*

Sometimes, phrases can contain adjuncts which are not pronounced as part of the sentence, and have no overt marking. Traces are one example of these so-called **empty categories**. Another example is *PRO*, a pronoun which is sometimes implicit in a sentence [11]. Consider the sentences:

(3) John wants to eat.

(4) John is considering whether to talk.

(5) To run away would be cowardly.

Each of these sentences contains a verb that does not have an overt subject: in sentence (3), it is the verb "to eat;" in sentence (4), it is the verb "to talk;" and in sentence (5), it is the verb "to run away." Each of these verbs actually has a non-overt pronoun, known as *PRO*, as its subject. Like traces, *PRO* is not pronounced. However, it is present in the syntax tree, and can sometimes be detected by the timing of speech or by other means [26]. The positions of *PRO* in sentences (3), (4), and (5) are:

(3′) John wants *PRO* to eat.

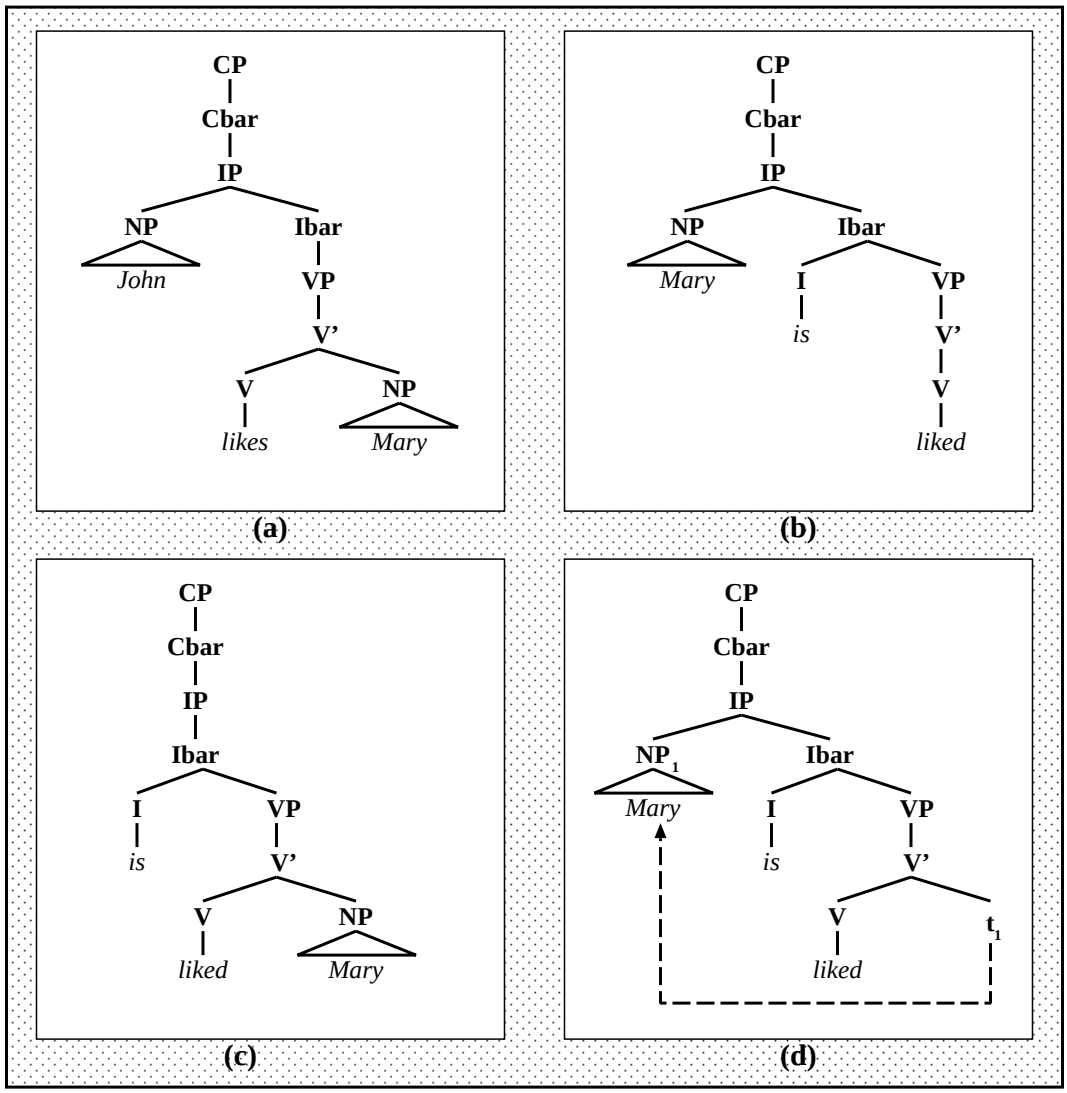

Figure 2-4: Illustration of Syntactic Traces. *(a)* The syntax tree representing the sentence "John likes Mary." *(b)* Simplified version of the syntax tree representing the sentence "Mary is liked." This syntax tree does not include traces. *(c)* The deep structure of the sentence "Mary is liked." This structure encodes the fact that "Mary" was originally an adjunct of the verb "like." *(d)* The surface structure of the sentence "Mary is liked." The trace indicates the original position of the noun phrase "Mary," and the co-indexing of the trace and the noun phrase indicate that they corefer. Arrows are also sometimes used to indicate coreference.

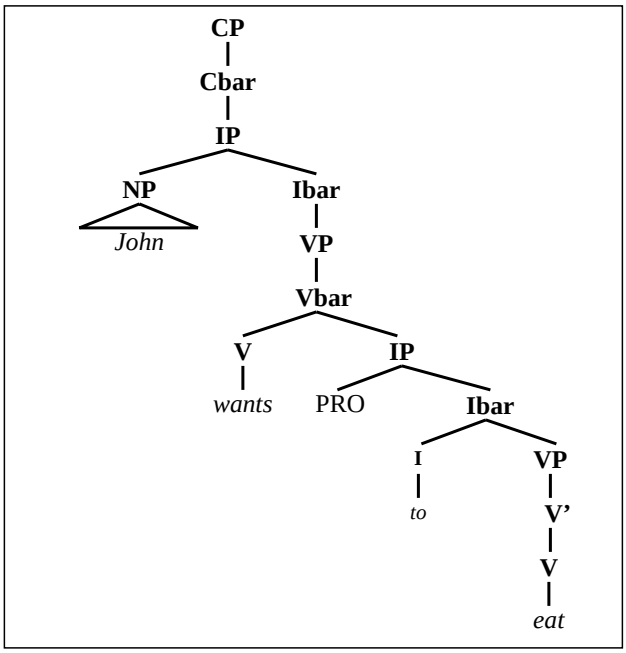

Figure 2-5: The syntax tree representing the sentence "John wants to eat." In this sentence, *PRO* acts as the subject of the verb "to eat." *PRO* is a non-overt pronoun.

(4′) John is considering whether *PRO* to talk.

(5′) *PRO* to run away would be cowardly.

The syntax tree for (3) is also given in Figure 2-5.

2.3.2 Arguments and Modifiers

Syntactic phrases can take two distinct types of adjuncts: arguments and modifiers. Intuitively, **arguments** are necessary to complete the meaning of the verb, while **modifiers** just provide auxiliary information [11]. Consider the verb phrase:

(6) forcefully **gave** an apple to Mary on Wednesday

where the head is written in bold face, and each adjunct is underlined. In this phrase, the adjuncts "an apple" and "to Mary" are arguments, since they complete the meaning of the verb "to give." In contrast, the adjuncts "forcefully" and "on Wednesday" are modifiers, since they simply provide auxiliary information. Each head can take a fixed number of arguments, and the roles that these arguments can fill is dependent on the head. Most heads take from zero to four arguments, and very few heads take

more than six.

Another way to the express the difference between arguments and modifiers is that modifiers always give the same type of auxiliary information, independent of the head that they are attached to; but the type of information arguments give about the head depends directly on their head. For example, consider the sentences:

(7) John **watched** Mary in the park.

(8) John **greeted** Mary in the park.

The semantic role played by the adjunct "Mary" in (7) is very different from its role in (8). In contrast, the adjunct "in the park" serves the same purpose in both sentences: it identifies the location of the event being described. Thus, in these sentences, "Mary" is an argument of its verb, while "in the park" is a modifier.

In addition to the intuitive and functional distinctions between arguments and modifiers, there are syntactic reasons to believe that they are different. For example, arguments can receive case, while modifiers can not [11].

The distinction between arguments and modifiers is important when trying to determine how an adjunct relates to its head. If an adjunct is a modifier, then the way it relates to its head is implicit in the type of modifier: location modifiers always specify the location of an event, time modifiers always specify the time of an event, and size modifiers always specify the size of an object. However, the relation of an argument to its verb cannot be derived from the argument itself; instead, it must be explicitly encoded.

2.3.3 Thematic Roles

The most common way of encoding the relation between a head and its arguments is through the use of thematic roles. **Thematic roles**, or **θ-roles**, are symbolic labels for each of the different relations an argument can have with its head. Some commonly used thematic roles are listed in Figure 2-6.

There has been some dispute about what set of labels should be used for thematic roles [11]. However, the exact set of labels used is not important to an IR system, as long as every argument of a head is assigned a unique thematic role. This ensures

Thematic Role	Example
BENEFACTIVE	John **gave** Mary a book.
THEME	John **gave** Mary a book.
EXPERIENCER	Murder mysteries **please** John.
SOURCE	John **took** the book from Mary.
DOMAIN	The **president** of France
THEME	**proud** of the pig

Figure 2-6: Commonly Used Thematic Roles. The argument whose θ-role is specified is underlined, and its head is written in bold face.

that, when two phrases are matched against each other, the proper arguments can be compared to each other, even if they occur in different positions, as in the following sentences:

(9) John gave Mary the ball in the park.

(10) John gave the ball to Mary in the park.

Often, linguists use thematic roles to encode the relationship between a head and a phrase that is not technically an argument of the head [11, 21]. For example, in the sentence "John will give the book to Mary," "John" is said to fill the AGENT thematic role of "give," even though it is an argument of "will," not "give." Although this practice of assigning theta roles to phrases that are not arguments is very common, recent research suggests that it may not accurately reflect the way humans process language [17].

2.3.4 Alternations

Often, one semantic notion can be expressed using a number of **alternations**, or syntactic variants [18, 16, 11]. In this case, the thematic roles assigned to the arguments are the same in each sentence, even though the location of the argument is different [11]. For example, consider the following sets of sentences:

(11) John gave the ball to Mary.
(11′) John gave Mary the ball.
(11″) Mary was given the ball by John.

(12) I took John's hand.
(12′) I took the hand of John.

(13) We threw the ball around in the park.
(13′) In the park, we threw around the ball.

In each of these sets of sentences, the same basic idea is expressed, using the same words, but different word orders are used. In order for a semantic information retrieval system to successfully decide when a sentence understands a question, it must be capable of determining which thematic roles should be assigned to which arguments.

2.3.5 Verb Classes

Many different verbs allow the same types of alternations. For example, the verbs "throw," "toss," and "smash" can undergo many of the same alternations [18]. Building on this intuition, Beth Levin has found evidence that English verbs can be divided into a relatively small number of distinct classes, each of which defines which alternations are possible for its verbs [18].

Each of these **verb classes** specifies a fixed set of thematic roles, which each of the verbs in the class can take. Additionally, it specifies the alternations which the verb and its arguments can undergo. A semantic IR system can greatly reduce the possible ways that the verb's thematic roles can be assigned to its arguments if it can decide which class the verb belongs to. Often, this can be accomplished by simply looking the verb up in a table.

However, a single verb often belongs to multiple verb classes. For example, the verb "take" most likely belongs to different verb classes in each of the following sentences:

(14) John took Mary to the store.
(15) John took the apple.
(16) John took a walk along the river.

This multiplicity of verb classes for a single verb occurs most commonly when one word has multiple meanings. Detecting the correct verb class requires more advanced techniques than simple table look-up. However, it is possible that the verb class information itself will actually allow us to disambiguate the verb, if the sentence's structure is only consistent with one of its possible verb classes' alternations.

Beth Levin restricted her discussion of word classes to verbs. But in principle, any lexical head can assign a thematic role. The notion of verb classes can be easily extended to cover all lexical heads. Each lexical head is assigned a **semantic class**, which describes the set of thematic roles that the head can assign, and the set of alternations that it can undergo. Note that words with different parts of speech might share the same semantic class. For example, it might be reasonable to assign the same semantic class to "destroy" and "destruction" in the following phrases, since it assigns the same set of thematic roles to its arguments:

(17) John destroyed the city
(18) John's destruction of the city

2.3.6 Predicate Logic

The relationships between the heads of a sentence and their adjuncts gives a great deal of information about the semantic content of the sentence. However, these relationships alone fail to capture what we think of as the "meaning" of the sentence. For example, as mentioned above, the subject of a sentence is not an argument of its verb, even though it clearly serves to complete the meaning of the verb. Another example of adjunct-level relationships failing to capture what we think of as the "meaning" of a sentence is the relationship between determiners, such as "every" or "the," and the nouns they modify.

One approach to finding all these relations is based on predicate logic [13]. Predicate logic assumes that, to each verb, there corresponds a predicate that describes that verb. The arguments of the predicate include both the proper arguments of the verb, and any **external arguments**, or phrases which serve to complete the meaning of the verb even though they are not proper arguments. For example, the sentence "John gave the book to Mary" might be expressed by the predicate:

$$\texttt{give}(\textit{John}, \textit{book}, \textit{Mary})$$

This predicate form is useful for the task of information retrieval, since compar-

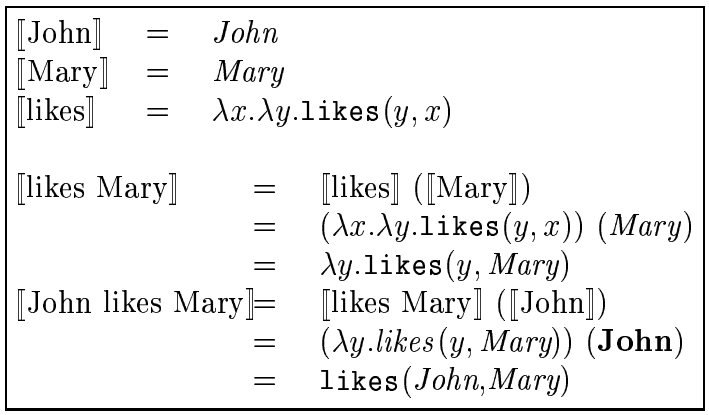

Figure 2-7: Example of Fregean Semantic Derivation. This figure illustrates the Fregean semantic derivation of the predicate corresponding to the sentence "John likes Mary." It makes use the $[\![\bullet]\!]$ operator, which finds the semantic value of any syntactic phrase. Functions are expressed using lambda-calculus. The first three lines give the definitions of the values of each of the words in the sentence. Note that the value of **likes** is a function. These values are then combined via functional application to yield the final predicate, **likes**(**John**,**Mary**).

ing the question's predicate to the predicates of the corpus's sentences can give us information about which sentences might answer the question.

2.3.7 Fregean Semantics

A common way of deriving the predicates that describe a sentence is known as **Fregean Semantics**, based on the work of Gottlob Frege in the late nineteenth and early twentieth centuries [13]. Fregean Semantics is based on the notion that the derivation of predicates from sentences is best modeled with the use of functional application. Each node in a syntax tree is given a value, which can be an object, a predicate, or a function. The value of each node is simply derived by applying the value of the head (which should be a function) to the values of each of the arguments. As an example, the derivation of the sentence "John likes Mary" is given in Figure 2-7.

Fregean Semantics allows syntactic trees to be compositionally combined to yield logical expressions containing the predicates that describe the sentences. The types of transformations allowed by Fregean Semantics can be quite complex, and are powerful enough to account for many linguistic phenomena, including quantifiers (such as "every") and pronouns [13].

However, a number of obstacles make it difficult to use Fregean systems for information retrieval. First, whenever a Fregean system is designed to handle anything beyond very simple sentence, the functions used to represent words become very complex. Indeed, no one has been able to develop a Fregean system that can can assign a set of predicates to most English sentences. Second, the Fregean system requires that each individual word be given its own definition. This fact makes it very difficult to build a high-coverage Fregean system. Finally, Fregean systems do not have any method of recovering from inputs that they don't know how to process. For example, if a Fregean system is given words that it does not have definitions for, or if words are used in ways that the system was not designed to handle, the system will simply fail. Until Fregean analyses of language become more advanced, it will be difficult to incoroperate them into information retrieval systems.

2.3.8 Semantic Trees

A more practical approach for an information retrieval system might be to use a hybrid between predicate logic and syntax trees. This hybrid would retain much of the original structure of the sentence, but would include both internal and external arguments of a head as its children.

To that end, I propose a simple structure, which I will call a semantic tree. A **semantic tree** is a tree-structured representation of a sentence, composed of **semantic phrases**, where each semantic phrase consists of a single lexical head, its modifiers, and all of its internal and external arguments. Each of the phrase's arguments is labelled with its thematic role. An example semantic tree is given in Figure 2-8.

2.4 START

The design of SQUIRE was heavily influenced by the START system, which answers questions by comparing them to individual assertions [16, 15]. The START engine is quite successful at answering questions about a natural language reference source. It can respond to a wide range of questions about several domains, including MIT's

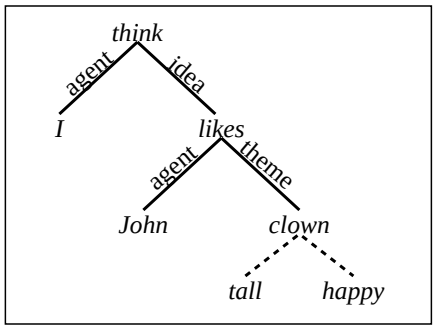

Figure 2-8: Semantic Tree for the sentence "I think John likes the tall, happy clown." Each semantic phrase consists of a lexical head and its arguments and modifiers. Arguments are connected with solid lines, labeled by their thematic role. Adjuncts are connected with dotted lines.

Artificial Intelligence Lab, movies, and world geography [16, 15]. Abstractly, START works by examining a question and matching it against each declarative sentence that it has been told. If it finds a sentence that answers the question, it will return that sentence, and the result of matching the sentence is running the procedure.

Internally, START converts each declarative sentence into one or more **links**, each of which encodes a basic relationship from the sentence, in the form of a subject, relation, and object. For example, the sentence "John eats the green apple" will be encoded with the links `<john, eat, apple-1>` and `<apple-1, is, green>`. These links are then stored in START's knowledge database.

When START is given a query, it converts the query into links, and tries to match these links against those stored in the database. A query link **matches** a database link if the database link implies the question link. For example, "John ate an apple" implies that "John ate," and "John gave Mary the ball" implies that "John gave the ball to Mary." If each of the query's links match a database link, then START uses them to answer the question. Otherwise, START reports that it doesn't know the answer to the question.

If a query link and a database link are identical, then clearly they should match. However, non-identical links can also match each other, under two circumstances. First, two different links can encode syntactic alternations of the same form. Second, the relationship encoded by one link might imply the relationship encoded by another

link. In either of these cases, the links should match. To handle both of these cases, START uses **S-rules**, each of which states that one class of links implies another class of links. If the application of START's S-rules to a database link results in a query link, then the two links match.

2.4.1 The Structure of Links

In order to process sentences, START needs to translate them into a simpler internal representation. START parses each sentence into **links**, each of which encodes some sub-part of the sentence. Currently, each link consists of a relation, a subject, an object, and a history:

- The **relation** is usually derived from the main verb of a clause. It is a symbol, such as "eat," "run," or "is-a."
- The **subject** is usually derived from the subject of a clause. It is either a symbol (usually representing a concrete object) or a link.
- The **object** is usually derived from a direct object of a clause. Like the subject, the object can be either a symbol or a link.
- The **history** contains any other information about the sentence, including adverbs, verb tense, and indirect objects.

Note that START decides what to put in the subject or object positions of links purely on the basis of their syntactic position. For example, in the sentence "I told John," "John" is the object; but in the sentence "I told the fact to John," "the fact" is the object. Thus, in different sentences with the same verb, the subject and object can have very different relations to the verb.

Although the links derived from declarative statements and from questions have the same general structure, there is an important difference between them: query links can contain variables, whose values are the answer to the query. For example, the link representing the question "who does john like?" is `<john, like, ?x>`, where

`?x` is a variable. The variable `?x` will match any value, and the answer to the question will be the value that `?x` matches.

Chapter 3

Exploring Semantic IR

Most researchers agree that it should be possible to use semantic information to improve the performance of information retrieval systems [12, 28, 2]. However, it is unclear which of the many possible methods of incorporating semantic information into an information retrieval system will yield substantial improvements [14, 24]. In order to explore which of these methods seem most promising, I designed and built SQUIRE, a Semantic Question-Answering Information Retrieval Engine. SQUIRE is intended to be a test-bed which can be used to explore the effects of a variety of semantic improvements to traditional IR systems.

3.1 SQUIRE's Task

The task I chose for SQUIRE was to find the individual sentences which answer a natural language question from a large corpus (on the order of 1 million sentences). The types of questions given to SQUIRE should be specific questions, that can be answered by single sentences. Some examples of such questions are:

- What do whales eat?
- What is the capital of Norway?
- Where are factories located in Honolulu?
- What are the chief crops grown in Guinea-Bissau?

- Who invented football?

SQUIRE is not optimized to answer general questions that ask for information about a single entity, such as:

- Tell me about monkeys.
- What are missionaries?
- Who was Shah Jahan?

Since these questions do not contain any semantic *relationships*, they would best be answered using traditional information retrieval techniques. However, ideally, SQUIRE should perform as well as traditional systems when faced with such questions.

I chose the task of sentence-level IR because it is a task at which traditional IR performs poorly, and a task for which semantic IR techniques show promise. Sentence-based IR is an important task, and is often preferable to article-based IR, since it is much easier to skim several sentences to find the answer to a question than it is to skim several articles. Additionally, the cost of returning incorrect sentences is decreased, since the user can usually determine immediately if a sentence returned does not answer their question. In contrast, returning incorrect articles often requires the user to skim most of the article before they can determine that it does not answer their question.

3.2 Design Criteria

Although precision and recall provide simple metrics for measuring the performance of a system, there were many other criteria that SQUIRE was designed to satisfy. These criteria can be divided into properties of the system's question-answering behavior (response criteria), and properties of the system itself (system criteria).

3.2.1 Response Criteria

- *Wide Coverage:* Often, semantic systems are built to perform very well in a limited domain, but perform poorly outside that domain. However, much of

the usefulness of information retrieval systems comes from the fact that they can be applied to novel situations. Although SQUIRE makes use of domain-specific information, I designed SQUIRE to behave reasonably when operating outside of known domains, such that it can always perform at least as well as traditional IR systems. For example, if SQUIRE can't determine the structure of a sentence, it still tries to index the constituent pieces that make up the sentence.

- *High Precision:* The amount of textual information available from electronic sources, including the Internet, has been increasing at a phenomenal rate for the past 10 years. The information necessary to answer almost any question is most likely available from many different sources. But users are usually only interested in retrieving one, or perhaps a few, answers to a question. Thus, as the amount of information available grows, precision becomes more important than recall. I therefore designed SQUIRE to primarily maximize precision, and only secondarily maximize recall.

- *Linguistic Normalization:* A single idea can be formulated using many different words and constructs. This linguistic variation can make it difficult to decide whether a sentence answers a question. This task becomes much easier if all linguistic variants of the same idea are reduced to a normalized form. I therefore designed SQUIRE to detect linguistic variation and produce normalized representations of its input whenever possible.

- *Rapid Query Response:* It is unreasonable to ask users to wait more than a few seconds for a response to their question. Since SQUIRE indexes such a large amount of information, it is important that it not have to search through all of that information every time it responds to a query. For that reason, SQUIRE was designed to provide rapid responses to questions, even when indexing very large corpora.

3.2.2 System Properties

- *Modularity:* SQUIRE is designed to explore the impact of a number of different semantic improvements to traditional IR techniques. In order to place minimal restrictions on the types of improvements that can be tested within the SQUIRE framework, I used a very modular design. Each improvement is implemented within its own self-contained module. These modules communicate with each other through a simple, well-specified interface. This allows SQUIRE to be easily extended to test further semantic improvements. Additionally, this allows SQUIRE to test the interactions between various potential improvements.

- *Domain Extensibility:* In order for an information retrieval system to effectively extract the meaning from a sentence, it must have information about the syntax and semantics of the language. In particular, it must have specific information about words, types of words, and syntactic constructs. However, the task of encoding information about all of the words, types of words, and constructs in English is a daunting task. Therefore, I designed SQUIRE to ensure that the new information about specific words and constructs could easily be added. In addition, following the criterion of wide coverage, I designed SQUIRE to use reasonable default information whenever it encountered unknown words, word types, or constructs.

- *Simplicity:* A simple model is easier to understand, reason about, implement, and maintain.

- *Efficiency:* SQUIRE's performance is affected by a large number of parameters, whose values are best determined by experimentation. Changing some of these parameters requires the entire index to be rebuilt. To allow rapid exploration of the effects of changing these parameters, SQUIRE was designed with efficiency in mind.

Chapter 4

SQUIRE: A Semantic Information Retrieval Engine

The SQUIRE system is divided into two modules: a **parser** and a **database**. The parser converts sentences and questions into weighted sets of **relations**, each of which partially encodes a semantic concept. The database maintains an index of the weighted relations that represent a corpus. When the user asks SQUIRE a question, the parser converts it into a weighted set of relations, and the database then uses those relations to retrieve the sentences most likely to answer the user's question.

4.1 Weights

SQUIRE makes extensive use of weights. These weights indicate SQUIRE's confidence that a given piece of information is useful. This confidence is derived from a number of sources, including:

- An overall valuation of the original source of the information. For example, information derived from an encyclopedia might carry a higher weight than information derived from an unknown web page.
- The likelihood that the information correctly represents the original source. When SQUIRE performs a transformation that might be incorrect or inappro-

priate, it adjusts the weight of its output accordingly.

- The number of independent sources that converge on the same information. For example, if SQUIRE performs multiple transformations that give the same result, then the weights resulting the individual transformations are combined.

- The uniqueness of the information. Information which is present in many sources is a given lower weight than information that is present in only a few sources.

Weights are represented as floating point numbers in the range $[0, 1]$. Weights can be combined in a number of ways, but they are most commonly combined with the *and* and *or* operators:

$$w_1 \ \textit{and} \ w_2 = w_1 \times w_2$$

$$w_1 \ \textit{or} \ w_2 = 1 - ((1 - w_1) \times (1 - w_2))$$

Two weights are combined with *or* when they represent two different ways a piece of information might be useful. In particular, if multiple independent sources converge on the same information, then the weights of the individual sources will be combined with the *or* operator.

Two weights are combined with the *and* when they are independent valuations of the same piece of information. The *and* operator is used by parser layers to combine the weights of their inputs with their own valuation of the information.

4.2 The Parser

The parser is responsible for converting strings of words into relations. It consists of a set of **parser layers**, each of which transforms an input to produce an output. These transformations serve to extract the semantic content of the input, to reduce its linguistic variation, and to convert it to a form which will allow for easy indexing.

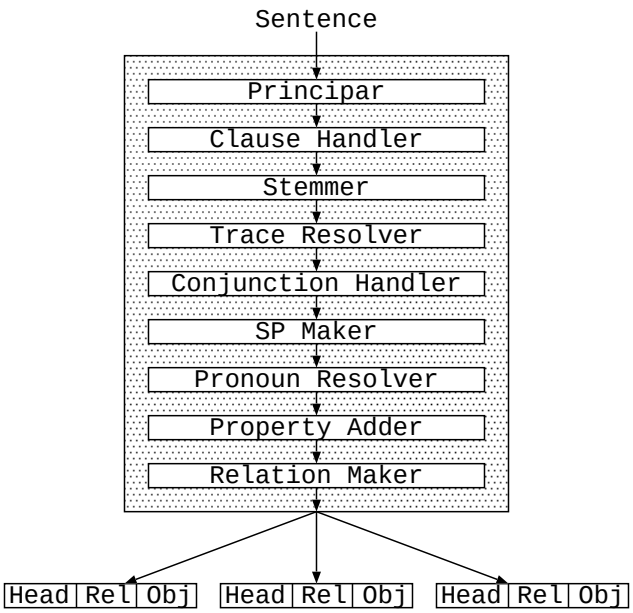

Figure 4-1: Example Parser. This parser is constructed by attaching a number of parser layers in series. The weighted output of each layer feeds into the input of the next layer.

Each parser layer transforms each individual weighted input into zero or more weighted outputs. Weights of the outputs reflect both the original weight of the input, and the effect of the transformation on the weight.

The parser is constructed by attaching a number of parser layers in series, such that the weighted outputs of one layer feed into the next layer. During indexing, the original input to the parser is simply the sentences to be indexed, weighted by the valuation of their source; the final output is the weighted set of relations to be indexed by the database. During question-answering, the input to the parser is the questions to be answered, with weight 1, and the output is the weighted set of relations to be looked up in the database. An example parser, constructed out of parser layers, is shown in Figure 4-1.

4.2.1 Classifications of Parser Layers

Parser layers can be divided into two basic classes, based on the relation between their input representation and their output representation:

- **representation-shifting layers** input information in one representation, and transform it into another representation.
- **source-to-source layers** use the same representation for both their input and their output.

Parser layers can be further divided into four subclasses, based on how they transform their input:

- **one-to-one layers** transform each input into exactly one output. Often, one-to-one layers do not modify the weight of their input.
- **Splitters** are used to split complex structures into several simpler structures. Splitters produce one or more outputs for each input. The weights of the outputs are generally equal to the weight of the input.
- **one-to-many layers** are used when SQUIRE cannot determine which of several possible representations best encodes an input. In this case, the layer outputs all possible representations, and each representation's weight is adjusted by SQUIRE's confidence that it is correct.
- **Filters** selectively remove some class of inputs, and pass on the remaining inputs. Filters generally do not modify the weight of their input. Filters can be used to remove items that do not have sufficient weights.

4.2.2 Advantages of the Layered Architecture

Simplified Testing and Prototyping

Dividing the parser into a set of modular layers, each of which has a simple interface, has a number of advantages. First, new semantic processing techniques can be easily tested by simply adding new layers to SQUIRE's parser. Second, the interactions between layers can be examined by running tests while varying which layers are used, and the order that they are used in. This is especially true of source-to-source layers.

Third, different algorithms for accomplishing a semantic task can be easily compared by encapsulating each in a separate module, and examining how the system's behavior changes when the modules are swapped.

Graceful Degradation

The fact that every piece of information in SQUIRE is assigned a weight allows a parser layer to process an input even when it is unable to reliably determine the proper output. For example, consider the sentence:

(19) Mary likes the man on the roof.

A parser layer might have difficulty deciding between the following interpretations of (19):

(19′) Mary likes the man who is on the roof.

(19′) On the roof, Mary likes the man.

In this case, it can simply output all of the possible outputs, each weighted by the the parser layer's estimate of the probability that it is a correct transformation. Often, incorrect interpretations will be given low weights by subsequent parser layers, or completely eliminated by filter layers. Thus, even if an incorrect interpretation of an original sentences makes it into the database, it will be given a low weight.

Cognitive Underpinnings

In addition to having useful prototyping properties, the division of the parser into modular layers may provide a good model of human language processing. Research in cognitive science suggest that humans process many aspects of language in a number of simpler steps, with simple interfaces between those steps. Although the exact set of steps used by the SQUIRE system are doubtless very different from those used by humans, the general architecture may allow us to examine some of the effects of processing language in a series of discrete steps. Additionally, as we come to understand more about the human language processing faculty, we may find new ways to improve the layered model I used in SQUIRE. For example, the human language faculty has inspired me to consider ways of allowing information to flow in

both directions between layers, allowing information from later layers to affect earlier ones.

4.2.3 Parsing in Parallel

When humans process sentences, they seem to consider the most promising interpretations first, and to only process the less promising interpretations if the more promising ones are eliminated for some reason. For example, when hearing sentence "The man pushed by John was named Harry." most people first interpret the words "the man pushed by John" to mean that the man was doing the pushing. However, upon encountering the phrase "was named Harry," this interpretation is eliminated, and they must go back and construct a new interpretation of the first part of the sentence.

In order to give SQUIRE similar behavior, I placed small buffers between each of the parser layers. These buffers sort the outputs of the prior layer, and feeds them into the next layer in descending order of weight. Thus, each layer processes the information with the greatest weights first.

In addition, I allowed each parser layer to run in parallel threads, and to apply back-pressure to the layers before it. Each layer would process its inputs until its output buffer was full. At that point, it would only continue processing if the output buffer shrank. Thus, less promising interpretations never needed to be computed by prior layers if more promising interpretations were available.

Unfortunately, the overhead of running each parser layer in parallel threads exceeded the benefit, and so I disabled this feature. However, I believe that this feature would still be useful if any parser layers were designed that generated a large number of outputs.

4.3 Representations

Before discussing the individual parser layers that make up SQUIRE's parser, I will describe the representations used to communicate between the layers. SQUIRE uses

four distinct representations to encode the semantic content of sentences and questions:

- Strings
- Syntax Trees
- Semantic Trees
- Relations

By transforming sentences and questions to this sequence of representations, I remove linguistic variation and normalize the input to a form suitable for indexing.

4.3.1 Strings

The original input for SQUIRE is a sequences of characters. This string is first broken up into individual sentence. Each sentence consists of a string of letters, spaces, and punctuation. Sentences are divided by terminating punctuation marks, such as periods and question marks.

4.3.2 Syntax Trees

A syntax tree is a tree-shaped structure consisting of phrases, words, traces, and antecedents.

- `phrases` represent syntactic phrases. Each phrase is assigned a type, such as "VP" (verb phrase) or "NP" (noun phrase), and has at least one child.
- `words` represent the words that make up the sentence. Each word has a surface form, and may have a stem. For example, the word "walking" would have the surface form "walking" and the stem "walk." Words may not have children.
- `traces` represent the syntactic traces that result from movement of syntactic phrases. Each trace is assigned a numerical identifier that specifies which antecedent was moved from the location of the trace.

- `antecedents` represent moved syntactic phrases. Each antecedent has a numerical identifier that specifies the antecedent's trace, and has the moved syntactic phrase as its single child.

PRO is represented as a special word. This simplifies the representation of syntax trees, and allows parser layers to treat it like any other pronoun.

Syntax trees can be classified according to the properties of their constituents. Syntax trees that contain antecedents and traces are **surface-structure syntax trees**. Syntax trees that contains no antecedents or traces are **deep-structure syntax trees**. Syntax trees whose words all have stems are **stemmed syntax trees**.

4.3.3 Semantic Trees

A semantic tree is a tree-shaped structure consisting of nested semantic phrases. A semantic phrases consists of a head, its arguments, its modifiers, and a set of properties.

- `heads` represent the words that head semantic phrases. Each head consists of a surface form, a stem, and a semantic class. The semantic class of a head need not be specified if it is not yet available.

- `arguments` represent the arguments of a semantic phrase. Each argument consists of a semantic phrase and a thematic role. The thematic role of an argument need not be specified if it is not yet available.

- `modifiers` represent the modifiers of a semantic phrase. Each modifier consists of a single semantic phrase.

- The `properties` of a semantic phrase are used to keep track of a number of features which semantic phrases might have. These features can include syntactic information, such as the part of speech of the head, the prepositions used by prepositional phrases, and the determiners used by nouns. They can also

Semantic Head	Semantic Class	Adjunct	Relation Type
see	observational verb	I	agent
see	observational verb	give	observed-object
give	transference verb	John	agent
give	transference verb	book	transferred item
give	transference verb	Mary	recipient

Figure 4-2: The relations used to index the sentence "I saw John give a book to Mary." Note that all heads are in stem form, since semantic phrases are derived from stemmed syntax trees.

include semantic information, such as whether a noun is animate. These properties are currently used to decide what thematic roles the arguments should be assigned, although they could also be used for other purposes.

Semantic trees containing heads whose semantic classes are unspecified or arguments whose thematic roles are unspecified are called **underspecified semantic trees**. Semantic trees containing no heads whose semantic classes are unspecified and no arguments whose thematic roles are unspecified are called **full semantic trees**.

4.3.4 Relations

In order to allow partial semantic trees to be indexed and matched, they must be broken down into simpler component. SQUIRE therefore breaks semantic trees into a number of **relations**, each of which encodes the relationship between a single head and its adjunct. Each relation consists of:

- A semantic head
- The head's semantic class
- One of the head's adjuncts
- The thematic role or relation type that relates the head to the adjunct.

An example showing how relations relate to sentence is given in Figure 4-2.

Relations can also be used to encode relations whose semantic class and relation type are unknown. In this case, the semantic class and relation type are just left blank. This allows SQUIRE to index relations even if it doesn't have enough information

Semantic Head	Semantic Class	Adjunct	Relation Type
John	-	-	-
feed	-	-	-
bone	-	-	-
dog	-	-	-

Figure 4-3: The relations used to index the words in the sentence "John will feed a bone to the dogs." Note that only lexical words are indexed, since relations are extracted from semantic phrases, and semantic phrases contain no functional heads. Note also that all heads are in stem form, since semantic phrases are derived from stemmed syntax trees.

to determine their thematic role or relation type. Relations whose semantic class and relation type are unspecified are **underspecified relations**. Relations whose semantic class and relation type are given are **full relations**.

Finally, relations can be used to encode individual words. When relations are used in this way, only the head field is used; the other fields are all left blank. By indexing all of the words in the semantic tree, in addition to its relations, SQUIRE gains some of the benefits of traditional IR techniques. Figure 4-3 shows how the words of a semantic tree might be indexed.

4.4 Parser Layers

SQUIRE currently defines eleven parser layers, which can be combined in a variety of ways to give different parsing behavior. The parser layers defined by SQUIRE are listed in Figure 4-4, and individually described in the following sections.

4.4.1 The Principar Layer

The first stage in processing a sentence is constructing its syntax tree. A number of syntactic parsing programs, with various properties, are freely available for research purposes. I chose to used Dekang Lin's Principar program to construct syntax trees for SQUIRE. However, it would be a fairly simple matter to adapt SQUIRE to use a different parser, if a more appropriate parser were found.

Principar is is based on the "Principles and Parameters" model of syntax that has

Layer Name	Input	Output	Description
Principar Layer	String	Syntax Tree	Parses strings into syntax trees.
Stemmer	Syntax Tree	Stemmed Syntax Tree	Stems all words in a syntax tree.
Clause Handler	Syntax Tree	Syntax Tree	Splits modifying clauses off from syntax trees.
Conjunction Handler	Syntax Tree	Syntax Tree	Turns single syntax trees that contain conjoined phrases into several simpler syntax trees.
Trace Resolver	Surface-Structure Syntax Tree	Deep-Structure Syntax Tree	Moves antecedents back to their original location
Semantic Tree Maker	Stemmed Deep-Structure Syntax Tree	Semantic Tree	Turns syntax trees into semantic phrases.
Pronoun Resolver	Semantic Tree	Semantic Tree	Replaces pronouns with their antecedents.
Property Adder	Semantic Tree	Semantic Tree	Adds properties to semantic phrases, based on their head.
Synonymy Layer	Semantic Tree	Semantic Tree	Uses synonyms to generate synonymous syntax trees.
Theta Assigner	Semantic Tree	Semantic Tree	Assigns thematic roles to a semantic tree's arguments.
Relation Maker	Semantic Tree	Relation	Converts semantic phrases into relations.

Figure 4-4: SQUIRE's Parser Layers

been developed since the early 1980's. I chose to use Principar because it is fast, gives fairly accurate parses, and always returns its best guess at a parse. This last property is important, because it allows SQUIRE to index parts of a sentence even if Principar is unable to find the correct structure for the entire sentence. Unfortunately, the Principar program does not provide any information about how certain it is of its parse. Otherwise, we could use this information to modify the weight of the output.

4.4.2 The Stemmer

The Stemmer is a source-to-source, one-to-one parser layer that finds the stem of every word in a syntax tree. Currently, it uses a large table to find the stems of known words, but future versions of SQUIRE will most likely employ more advanced stemming algorithms.

4.4.3 The Clause Handler

Although the semantic phrase representation is capable of encoding most sentences, there are some constructions which are difficult to directly encode in semantic phrases. Before syntactic trees can be translated into semantic phrases, these constructions must be handled.

One construction that is difficult to represent with semantic phrases is the relative clause. A relative clauses is a type of subordinate clause used to modify a noun phrase. Consider the following sentences, where the relative clauses are underlined, and the nouns they modify are written in bold face.

(20) I saw **the box** that's sitting on the hill
(21) I met **the man** who you know
(22) **The song** I'm thinking of is very short

In each of these sentences, the noun in bold face fills two semantic roles. In sentence (20), "the box" is both the `EXPERIENCE` of the verb "saw" and the `AGENT` of the verb "sitting." In sentence (21), "the man" is both the `AGENT` of "met" and the `THEME` of "know." And in sentence (22), "the song" is the `IDEA` of "thinking" and the `AGENT` of "is."

Although these nouns fill two separate semantic roles for two separate verbs, the semantic phrase structure only allows a phrase to fill a single semantic role. Thus, these sentences cannot be directly encoded as semantic phrases. A simple solution is to split each of these sentences into two separate sentences. The separated sentences corresponding to sentences (20), (21), and (22) are:

(20′) I saw the box. The box is sitting on the hill.
(21′) I met the man. You know the man.
(22′) The song is very short. I'm thinking of the song.

Sometimes this transformation can distort the meaning of the original sentence. For example, consider the following sentence, and the two sentences that are derived by splitting it up:

(23) John does not think there is a man who is under the table.
(23′) John does not think there is a man. The man is under the table.

Although this distortion would be problematic for many semantic systems, such as inference engines, it generally does not interfere with the task of information retrieval. The reason is that the task of an information retrieval engine is not to directly answer a question, but only to return any sentences which are relevant to answering it. Even though sentence (23) does not directly answer the question "Is there a man under the table," it is relevant to the question, and should certainly be returned in response to it.

The Clause Handler is a source-to-source, splitter parser layer responsible for breaking relative clauses off into their own sentences. It takes syntax trees as its input, and for each syntax tree it inputs, it inputs one or more syntax trees representing the sentence and all of its relative clauses. Currently, the Clause Handler does not modify the weights of the syntax trees it processes.

4.4.4 The Conjunction Handler

Another construction which is difficult to directly encode with semantic phrase is the conjoined phrase. Conjoined phrases are two phrases combined by a conjunction, such as "and" or "or" to produce a single phrase. Consider the following sentences,

where each child of the conjoined phrases is underlined, and the head they modify is written in bold face.

(24) I **saw** the cat and the mouse.
(25) John **likes** Matthew but not Mary.
(26) Green and red **are** his favorite colors.

In each of these sentences, the two underlined phrases each fill the same semantic role of the bold-faced head. In sentence (24), both "the cat" and "the mouse" both fill the `EXPERIENCE` θ-role of the verb "saw." In sentence (25), "Matthew" and "not Mary" both fill the `THEME` θ-role of the verb "like." And in sentence (26), the `ASSIGNEE` θ-role is filled by both "red" and "green."

However, semantic phrases only allow a single phrase to fill a semantic role. This situation is analogous to relative clauses, where one phrase filled two semantic roles. Indeed, we can use the same solution to handle conjoined phrases: simply split each sentence into two simpler sentences. The separated sentences corresponding to sentences (24), (25), and (26) are:

(24′) I saw the cat. I saw the mouse.
(25′) John likes Matthew. John does not like Mary.
(26′) Green is his favorite color. Red is his favorite color.

Note that, as with relative clauses, this transformation does not always preserve the semantic meaning of the input sentence. For example, consider the following sentence, and sentences that are derived by splitting it up:

(27) John met Mary in the library or in the kitchen.
(27′) John met Mary in the library. John met Mary in the kitchen.

But as with conjunctions, this distortion of meaning should not interfere with the task of information retrieval, since the resulting syntactic phrases still contain the relationships that are relevant to the original sentence.

This analysis of simple conjoined phrases also applies to complex conjoined phrases, top-level conjoined phrases (i.e., conjoined phrases that are not arguments of any head), and appositives, as illustrated in the following examples:

(28) Monkeys eat eggs, fruit, grass, and insects.
(28′) Monkeys eat eggs. Monkeys eat fruit.
Monkeys eat grass. Monkeys eat insects.

(29) A bell rang out and the tower fell.
(29′) A bell rang out. The tower fell.
(30) The Eiffel Tower is located in Paris, the capital of France.
(30′) The Eiffel Tower is located in Paris.
The Eiffel Tower is located in the capital of France.

The Conjunction Handler is a source-to-source, splitter parser layer responsible for breaking sentences with conjoined phrases and appositives into simpler sentences. It takes syntax trees as its input, and for each syntax tree it inputs, it inputs one or more syntax trees representing every combination of conjoined phrases. Currently, the Conjunction Handler does not modify the weights of the syntax trees it processes.

4.4.5 The Trace Resolver

Before a surface-structure syntax tree can be converted into semantic trees, it should be restored to the deep-structure tree from which it originated. This deep-structure tree properly captures the sentence's semantic relationships. Converting a surface-structure tree to a deep-structure tree is a simple matter of replacing each trace with its antecedent, and removing the antecedent from its moved position. For example, the sentence:

(31) John was given a book

would be converted into:

(31′) was given John a book.

The Trace Resolver is a source-to-source, one-to-one parser layer responsible for converting surface-structured syntax trees into deep-structured ones. Currently, the Trace Resolver does not modify the weights of the syntax trees it processes.

4.4.6 The Semantic Tree Maker

The Semantic Tree Maker is a representation-shifting, one-to-many parser layer. It is responsible for converting syntax trees into underspecified semantic trees. It assumes that the syntax trees it is given are stemmed and deep-structured, and that they contain no relative clauses or conjoined phrases.

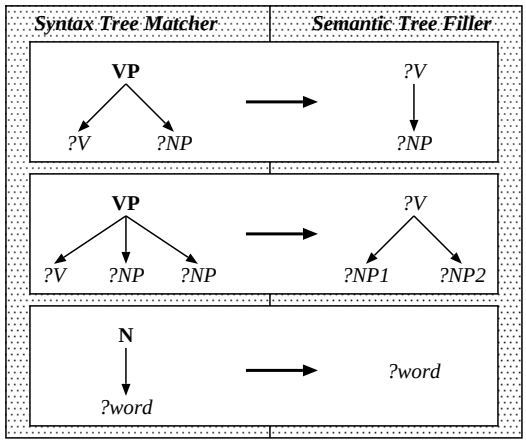

Figure 4-5: Example Templates for the Semantic Tree Maker. These templates are used to allow SQUIRE to convert syntax trees into semantic trees. Each template specifies that any syntax tree matching the template's syntax tree matcher can be replaced by the semantic tree obtained by filling in the semantic tree filler's variables.

In order to convert syntax trees into semantic trees, the Semantic Tree Marker uses a set of **templates**, which describe the relationship between syntax trees and semantic trees. The exact set of templates used depends primarily on phrase classifications used by the syntactic parser. Currently, the Semantic Tree Maker uses a set of templates designed to handle the syntactic structures generated by Principar. Several example templates are shown in Figure 4-5.

Each template consists of a syntax tree matcher and a semantic tree filler:

- A **syntax tree matcher** is a representation of a small piece of syntax tree. It is structurally identical to a syntax tree, except that it can contain variables in place of syntax phrases and words. A syntax tree matcher is said to **match** a syntax tree if some set of values could be filled in the for the matcher's variables to give the syntax tree. The set of values used to fill in the syntax tree matcher's variables are called the variables' **bindings**.

- A **semantic tree filler** is a representation of a small piece of a semantic tree. It is structurally identical to a semantic tree, except that it can contain variables in place of heads or adjuncts. Semantic tree fillers can be used to construct semantic trees by replacing their variables with values. This process is known

as **filling** the semantic tree filler's variables.

Every template specifies a valid way of transforming a piece of a syntax tree into a semantic phrase. In particular, the following set of steps can be used to convert a syntax phrase into a semantic phrase:

1. Find a template whose matcher matches the syntax phrase.
2. Match the template's syntax tree matcher against the syntax tree.
3. For every variable that stands for a word, construct the head consisting of the variable's binding.
4. For every variable that stands for a syntax phrase, find the semantic phrase that represents the variable's binding.
5. Fill the template's semantic tree filler with the values obtained from steps (3) and (4).

Note that step (4) is recursive: the entire process described above must be repeated to convert each of the variable's bindings, which are syntax phrases, into semantic phrases. This recursion will always terminate, since all syntax trees have finite depth.

Multiple Template Matches

Occasionally, more than one template might match a syntax phrase. This might happen, for example, if there are two possible interpretations of a phrase. In this case, each template is applied separately, to generate as many outputs as there are matching templates.

Allowing multiple templates to match a syntax phrase should be avoided when possible, since the number of outputs will grow exponentially with the number of ambiguous syntax phrases in the sentence. For example, if a sentence contains six ambiguous syntax phrases, each of which can be described by two templates, the Semantic Tree Maker will output 2^6, or 64 separate semantic trees.

Template Weighting

Each of the Semantic Tree Maker's templates can be assigned a weight, which represents the probability that the template describes a correct transformation. These weights might be used, for example, when two or more templates can match a syntax phrase, to distinguish the relative probability that each of the matches is correct. Whenever a template is used to construct a semantic phrase, its weight is combined with the syntax tree's weight, using the *and* operator.

Unmatchable Syntax Phrases

Sometimes, the Semantic Tree Maker may encounter a syntax phrase which does not match any of its templates. This usually happens when Principar is unable to find the correct syntactic parse of a sentence. In this situation, the Semantic Tree Maker uses the following heuristics to recover:

- If the syntax phrase has exactly one child, the Semantic Tree Maker tries to find a semantic phrase representing that child. If it is successful, it returns that semantic phrase. However, it decreases the weight of the phrase combining it via *and* with a fixed weight, to account for the fact that this transformation may be incorrect. Currently, that fixed weight is 0.8.

- If the syntax phrase has multiple children, then the Semantic Tree Maker returns a special value, "UNKNOWN," which represents the fact that the value of that semantic phrase is unknown. The Semantic Tree Maker then tries to find semantic phrases representing each child. If successful, these semantic phrases are returned separately. To account for the fact that this transformation may be incorrect, the weights of these phrases are decreased by combining them via *and* with a fixed weight. Currently, this fixed weight is 0.8.

Advantages of the Template Architecture

I chose to implement the Semantic Tree Maker using this template-based architecture for two reasons:

- *Domain Extensibility:* The template-based architecture is a natural formalism for describing the relationship between syntax trees and semantic trees. Thus, the construction of a set of templates is a fairly easy task. It is an easy matter to add new templates to account for new syntactic constructs. If the parser used by SQUIRE is changed, constructing an entirely new set of templates to handle the new parser could be done in a matter of hours.

- *Wide Coverage:* The template-based architecture allows the Semantic Tree Maker to transform many parts of a syntax tree, even if it does not have sufficient information to transform the entire tree. Thus, even difficult-to-process sentences can be partially indexed.

4.4.7 The Pronoun Resolver

Pronouns are a significant source of difficulty for the application of traditional IR systems to sentences. Although the pronouns carry semantic information for any human reading them, the traditional techniques just treat them as functional words, which don't contribute to the sentence's meaning at all. Clearly, if an IR system is to decide accurately when a sentence with pronouns answers a question, it must first determine the semantic values of the pronouns. This semantic value comes from the pronoun's **antecedent**, or the word that the pronoun refers to. Once an IR system has found the antecedent of every pronoun in a sentence, it can substitute copies of the antecedents in place of the pronouns. This substitution process is called **pronoun resolution**.

To get a preliminary idea of the possible benefits of pronoun resolution, I wrote the pronoun resolver. The task of the pronoun resolver is to find the antecedents of both overt pronouns (such as "he" and "her") and non-overt pronouns (such as "PRO"), and replace each pronoun with its antecedents. It is a source-to-source, one-to-many parser layer.

There are a number of linguistic theories, such as Centering Theory [10, 4], whose algorithms can find the correct antecedents of pronouns with a high degree of relia-

bility. However, when I designed the pronoun resolver, I first wanted to get an idea of how worthwhile a full-blown pronoun resolving engine would be. I therefore used a much simpler algorithm, which took only about six hours to implement. This algorithm was sufficient to demonstrate that pronoun resolution gives significantly better performance, and will most likely be replaced by a more advanced algorithm in the future.

The current algorithm used by the pronoun resolver assumes that the antecedent of a pronoun is the subject of the sentence containing the pronoun (unless the pronoun itself is the subject), or the subject of one of the two preceding sentences. Each of these possibilities is given a weight, to reflect the fact that these possibilities are not equally likely: the subject of the current sentence is given a weight of 0.5 (unless the pronoun itself is the subject); the subject of the previous two sentences are given weights 0.3 and 0.2. Although this algorithm is very crude, it was very easy to implement, and was sufficient to significantly improve the performance of SQUIRE.

4.4.8 The Property Adder

Before deciding which arguments of a head should receive which thematic roles, it is often useful to determine several properties of the arguments. Consider the following pair of sentences:

(32) The plane flew to Mexico.

(32′) I flew the plane.

The argument "plane" fills the *VEHICLE* thematic role in both sentences, even though it is the subject of (32) and the object of (32′). In order to properly assign the correct thematic roles to the arguments of "fly," we must know something about their properties. For example, if we know that "I" is animate, while "plane" is not, we can determine that "I" functions as the *AGENT* of "fly," while "plane" cannot. Furthermore, once we decide that "to Mexico" is a destination, we can assign it the *DESTINATION* thematic role. Of the remaining thematic roles that the verb "fly" takes, "plane" can only fill the *VEHICLE* role, since it is a concrete, non-animate, and

is not a destination.

It could be argued that in sentence (32), the presence of the preposition "to" serves to distinguish it from (32′), and that this distinction should be sufficient to allow an IR system to assign thematic roles. However, this argument would not apply to the similar sentence pair:

(33) The plane flew.

(33′) I flew.

where the objects of each sentence have been omitted.

The Property Adder parser layer uses a database of word properties to determine the properties of each argument. For each semantic phrase in a semantic tree, it looks up the properties of the phrase's head, and adds those properties to the semantic phrase. Currently, the database it uses is quite small, although it could be extended in the future. The Property Adder does not change the weights of any of the semantic trees it processes. It is a source-to-source, one-to-one parser layer.

4.4.9 The Theta Assigner

The Theta Assigner is a source-to-source, one-to-many parser layer. It is responsible for splitting each semantic phrase's adjuncts into arguments and modifiers, and for assigning thematic roles to each argument.

Like the Semantic Tree Maker, the Theta Assigner makes use of a set of templates. An example template is shown in Figure 4-6. These templates are used to encode the possible ways that thematic roles can be assigned to a semantic phrase's adjuncts. The templates are divided up according to the semantic class they describe: each semantic class is described by one or more templates, which encode the possible syntactic alternations that can be used by that semantic class. Each template consists of a semantic tree matcher and a thematic filler:

- A **semantic tree matcher** is a representation of a small piece of an under-specified semantic tree. It is structurally identical to a semantic tree, except that it can contain variables in place of semantic phrases. These variable are

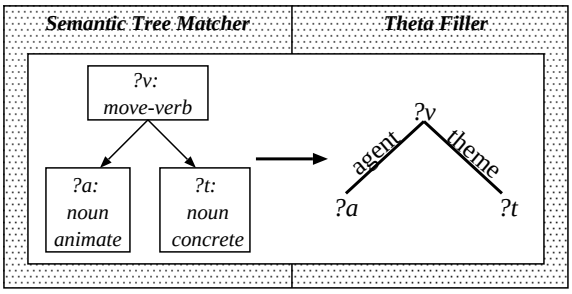

Figure 4-6: Example Template for the Theta Assigner. This example represents the type of templates used to allow SQUIRE to assign thematic roles to semantic trees. Each template specifies that any underspecified semantic tree matching the template's semantic tree matcher can be replaced by the full semantic tree obtained by filling in the thematic filler's variables.

each parametrized by a set of properties. A semantic tree matcher is said to **match** a semantic tree if some set of semantic phrases could be filled in for the matcher's variables, to give the semantic tree. Each value semantic phrase filled in for a variable must have properties consistent with that variable. The set of values used to fill in the semantic tree matcher's variables are called the variables' **bindings**.

- A **thematic filler** is a representation of a small piece of a full semantic tree. It is structurally identical to a semantic tree, except that it can contain variables in place of arguments, and it does not contain any modifiers. Thematic fillers can be used to construct semantic trees by replacing their variables with values. A set of modifiers can also be optionally added to the semantic tree. This process is known as **filling** the thematic filler's variables.

In addition to this database of templates, the Theta Assigner maintains a list of the possible semantic classes that each head may take. For example, most nouns can only take the semantic class *SIMPLE-NOUN*, while the semantic classes of the verb "get" might include *RECIEVE-ACTION* (in "he got the camera") and *BECOME-VERB* (in "he got better"). In this list, each of a word's possible semantic classes can be assigned a weight, to represent the relatively likelihood that a word will fill that semantic classes.

Each template specifies a valid way of assigning thematic roles to a semantic

tree's adjuncts. In particular, the following set of steps can be used to convert an underspecified semantic phrase into a full semantic phrase:

1. Find the possible semantic classes of the underspecified semantic phrase's head.
2. For each semantic class, find any template for that semantic class that matches the underspecified semantic phrase.
3. For each template found:
   (a) Convert the bindings of each of the template's variables from underspecified semantic phrases into full semantic phrases.
   (b) Fill the template's thematic filler with the values obtained from step 3a.

Note that step (3a) is recursive: the entire process described above must be repeated to convert each of the variable's bindings. This recursion will always terminate, since all semantic trees have finite depth.

Defining Semantic Classes

One of the most difficult aspects of building the Theta Assigner is the construction of a set of semantic classes. Even though a great deal of research supports the notion that words can be divided into semantic classes, no one has yet successfully constructed a complete, consistent set of semantic classes.

The semantic classes used to test the Theta Assigner were hand-constructed, and described only a small set of words. These classes were designed to help SQUIRE answer the test questions, in order to provide some indication of the potential gains that the Theta Assigner might provide. However, until a complete set of semantic classes can be developed, this layer will not provide any wide-coverage benefits.

4.4.10 The Synonymy Layer

One important type of linguistic variation that can prevent an information retrieval system from deciding that a sentence answers a question is synonymy. For example, consider the following question and answer:

(34) What do pigs eat?

(34′) Hogs feed on corn.

Although sentence (34′) answers question (34), it does not use the same set of words. Without some way of finding a words' synonyms, an information retrieval system will have no chance of finding answers like (34′).

As a preliminary investigation into the potential benefit of synonymy, I implemented a simple Synonymy Parser Layer. This layer uses a table of words and their synonyms to generate equivalent sentences. Each synonym in the table is given a weight, which reflects the probability that the synonym will accurately reflect the meaning of the original word. In order to reduce the chance that the Synonymy Layer will generate inappropriate synonyms, information about the semantic classes of words and their synonyms can also be included in the table. For example, "recieve" might be listed as a synonym of the *RECIEVE-ACTION* sense of "get," but not as a synonym of the *BECOME-VERB* sense.

When the Synonymy Layer encounters a sentence, it outputs that sentence, plus any other sentences obtained by replacing a word with one of its synonyms. Each of these sentence's weights is decreased according to the weights of the synonyms used. In order to prevent the generation of too many sentences, limits may be placed on the total acceptable decrease in weight of the original sentence.

Although the Synonymy Layer could be used when indexing a corpus, doing so would greatly increase the size of the index. A better approach is to only use the Synonymy Layer when processing questions. Thus, for example, if a user asks the question:

(35) What do pigs eat?

then the synonymy layer might add the questions:

(36) What do pigs feed on?
(37) What do hogs eat?
(38) What do hogs feed on?

The weights of each of these questions would be decreased slightly to reflect the fact that they may not accurately model the original question.

Currently, the the table of synonyms used by the Synonymy Layer is small, and its entries were entered by hand. These entries were designed to help SQUIRE answer the test questions, in order to provide some indication of the potential gains of adding a full-fledged synonymy layer. This small table will eventually be replaced by a larger table, constructed from an existing database of synonyms such as WordNet.

I also plan to extend the Synonymy Layer to allow the mapping between thematic roles of two synonyms to be specified. For example, "give" and "receive" could be listed as synonyms by specifying the following mapping between thematic roles:

give	**receive**
AGENT	*SOURCE*
BENEFICIARY	*AGENT*
THEME	*THEME*

4.4.11 The Relation Maker

The Relation Maker is a representation-shifting splitter that converts semantic trees into relations. It can operate in three basic modes:

- In **word extraction mode**, it outputs a single relation for each word in the semantic tree.
- In **underspecified relation extraction mode**, it outputs a single relation for every head/adjunct pair in the semantic tree. The semantic class of the head and the thematic role of the argument are left unspecified.
- In **full relation extraction mode**, it outputs a single relation for every head/adjunct pair whose semantic class and thematic role are known.

The Relation Maker can also operate in any combination of those modes. For example, it can be instructed to extract both words and full relations.

4.5 The Database

SQUIRE's database consists of a very large table of relations. For each relation in the database, this table contains a list of the locations of all the sentences containing that relation. For each sentence location, the database also contains a weight, indicating

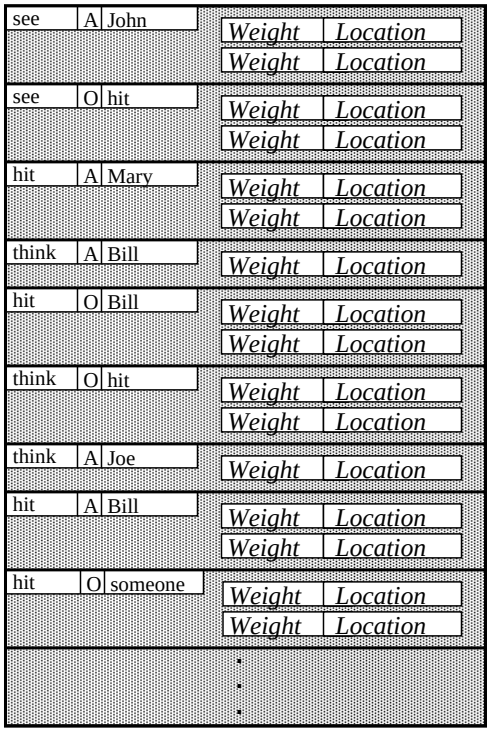

Figure 4-7: Database Structure. A database consists of a table of relations, and the locations of the sentences containing those relations. The weights associated with each location indicate the likelihood that the given location actually contains the given relation. In this figure, relations are written as tripples containing a head, a thematic role, and an adjunct. "`Weight`" and "`Location`" stand for actual weights and locations.

how strongly the location is associated with the relation. Figure 4-7 illustrates the structure of the database.

4.5.1 Indexing

When SQUIRE indexes a sentence, it first uses the parser to convert it into a set of weighted relations. For each of those relations, a pointer to the sentence is inserted into the database, along with the weight of the relation. This process is illustrated in Figure 4-8.

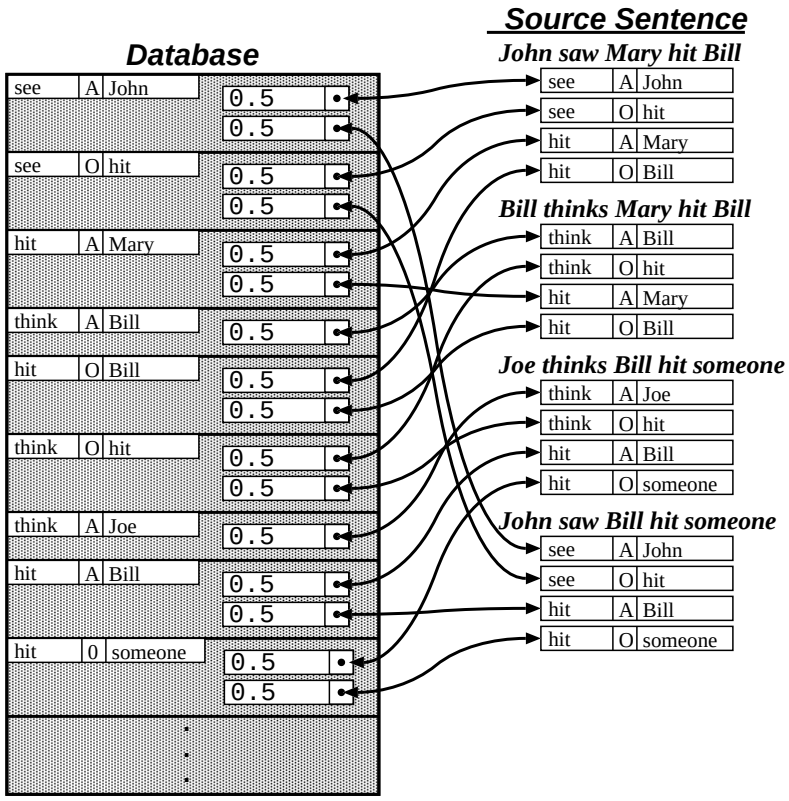

Figure 4-8: Database Indexing. First, the parser is used to convert a sentence into a set of weighted relations. For each relation in a sentence, an entry is added under the relation in the database specifying the location of the sentence it came from.

4.5.2 Retrieval

When a user asks SQUIRE a question, it first uses the parser to convert it into a set of weighted relations. The weights of these relations reflect SQUIRE's degree of certainty that the relations are actually relevant to the question. SQUIRE then looks up each relation in the database, to find the locations of each sentence that contains that relation. The weights of these locations are combined with the weights of the relations that generated them via the *and* operator. This accounts for the fact that a location is only relevant to a question if the location is relevant to the relation *and* the relation is relevant to the question. These weights are then divided by the number of times the relation appears in the corpus, to give higher weights to less common relations.

Once SQUIRE has generated a list of weighted locations for each relation, these weighted locations are merged. Whenever two or more relations independently generate the same location, the weights of those locations are combined via the *or* operator. This accounts for the fact that the location is relevant to the question if *either* of the relations indicates that it is relevant.

The process of retrieval is shown in Figure 4-9.

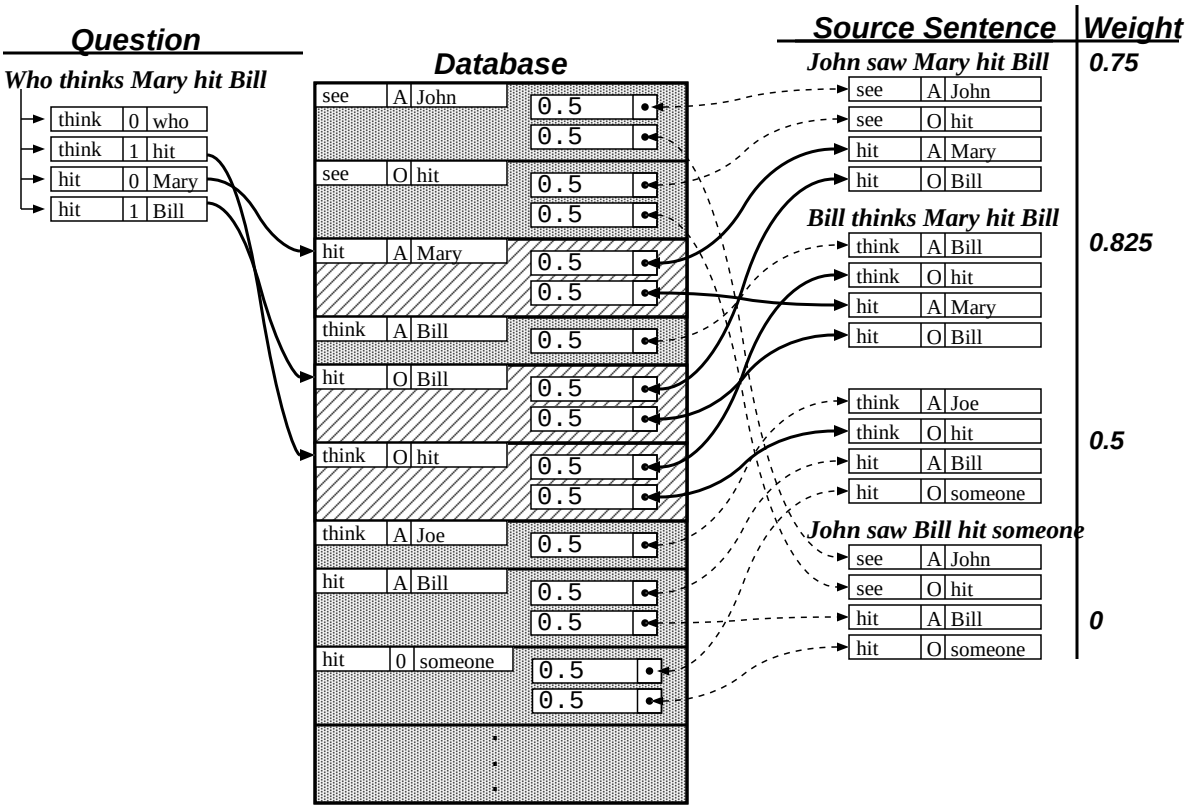

Figure 4-9: Database Retrieval. A database is first constructed by indexing the relations in the source sentences. When SQUIRE is given a query, each relation in the query is looked up in the database, and the resulting locations are returned, ordered by how likely they are to answer the question. The weights given for this example assume that all question relations have a weight of 1.0 and that all indexed relations have a weight of 0.5.

Chapter 5

Testing

The performance of the SQUIRE system was tested using the Worldbook Encyclopedia and a set of about 40 questions about animals and countries.

5.1 Test Materials

5.1.1 Test Corpus

The Worldbook Encyclopedia is a collection of about twenty thousand articles on a wide variety of topics [29]. The average article is about 500 words long, although article length varies widely. The Worldbook Encyclopedia was chosen because it consistently uses proper English sentences, and because it is a rich source of answers to common questions.

5.1.2 Test Questions

SQUIRE was tested with a set of 38 questions about animals and countries, including questions such as:

- What do seals eat?
- What do squirrel monkeys eat?
- Where do lions live?

- What is the capital of Benin?

The answers for the questions were found by hand, by reading all articles which seemed relevant to them. The average question was answered by four separate sentences, although the number of sentences answering a question varied widely, ranging from one to eighteen.

5.1.3 Test Indexes

I tested the SQUIRE system with twenty-eight different indexes, each constructed by using a different set of parser layers. These indexes explored each possible combination of the following five boolean parameters:

pro indicates whether the pronoun resolution layer is used.

syn indicates whether the synonymy layer is used.

word indicates whether individual words are extracted from the semantic phrases and included in the index.

ur indicates whether underspecified relations are extracted from the semantic phrases and included in the index.

fr indicates whether full relations are extracted from the semantic phrases and included in the index.

Note that at least one of *word*, *ur*, and *fr* must be true, since otherwise the index would be empty.

In all test cases, the Principar, Stemmer, Clause Handler, Conjunction Handler, Trace Resolver, Semantic Tree Maker, Property Adder, and Relation Maker parser layers were used.

5.2 Methodology

To test each configuration of parameters, I first constructed the necessary indexes. I then ran SQUIRE on each index, and asked it every question in the test set. For each

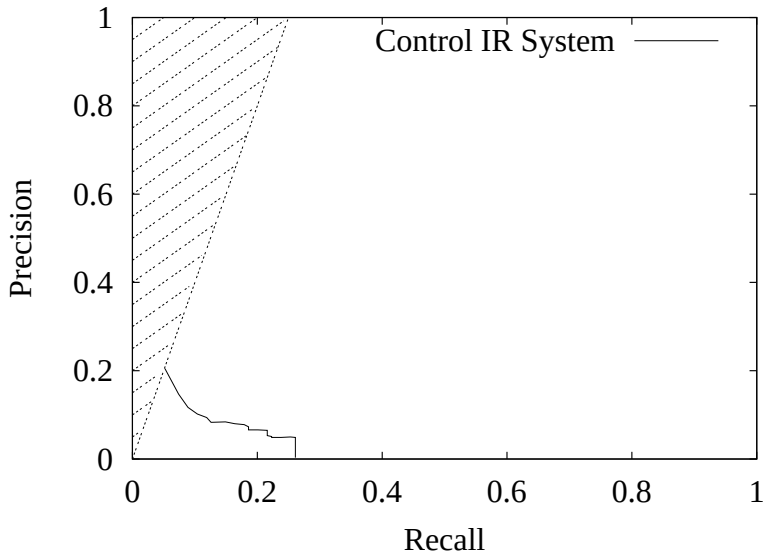

Figure 5-1: Precision-Recall Curve of the Control IR System

question, the first 300 responses were recorded, and compared against the known set of correct answers.

Precision-recall graphs were constructed by varying the number of SQUIRE's responses considered. For every n, $0 < n \leq 300$, I calculated a precision and recall value:

$$p_n = \frac{|A \cup B_n|}{|B_n|}$$

$$r_n = \frac{|A \cup B_n|}{|A|}$$

Where A is the set of sentences that correctly answer a question, and B_n is the first n sentences returned by SQUIRE. Note that this method does not give precision/recall pairs for any point where $p_n/r_n > |A|$, since that would imply that $|B_n| < 1$ (i.e., that SQUIRE returned less than one sentence). In all precision-recall graphs, the area where $p_n/r_n > |A|$ is shaded, to indicate that no data is available in this area.

5.3 Control

Precision-recall graphs are highly dependent on the exact test set used to generate them. I therefore ran my test set on an information retrieval system which uses traditional techniques to perform sentence-based IR. This system, which is a subset of the Strata IR engine, was developed by Jimmy Lin [19]. The precision-recall curve given by the control system is shown in Figure 5-1.

5.4 Results

Although each of the parameters interacts with the others to form the resulting precision-recall curves, it is informative to consider the individual effects of each parameter on the curve. I will therefore examine the overall effect of each parameter on the precision-recall curves independently, and discuss how these effects interact with each other.

5.4.1 Effects of the *word* parameter

Indexing the words from the corpus is very similar to performing traditional word-based IR. Figure 5-2 shows how their performance compares. SQUIRE slightly outperforms the control IR system, although neither achieves a precision higher than about 20%. I have not examined why SQUIRE outperforms the control IR system when it only indexes words, although it may result from differences in the weighting algorithms used by SQUIRE and by the control IR system.

Retrieval based on words gives low performance, although that performance can be augmented somewhat through the use of the pronoun resolution and synonymy layers, as shown in Figure 5-3. This configuration of SQUIRE gives fairly low precision, but its precision decreases gradually, which results in relatively good recall characteristics: the first 50 answers returned by SQUIRE contain over half of the correct answers.

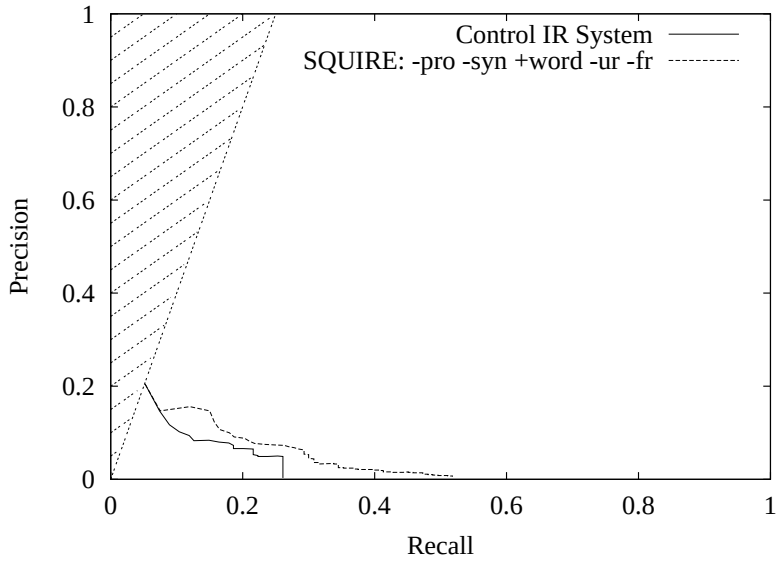

Figure 5-2: Comparison of Control IR System to SQUIRE Indexing Words

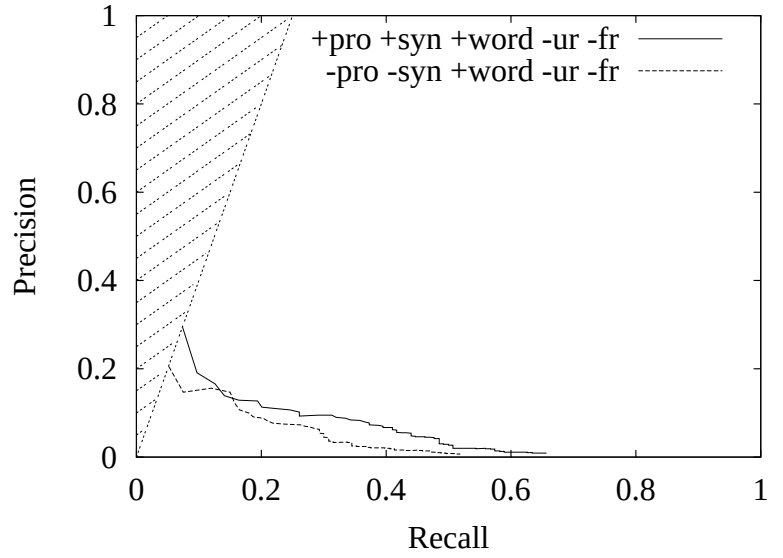

Figure 5-3: Precision-Recall Curve of SQUIRE Using the Pronoun-Resolution and Synonymy Layers to Index Words

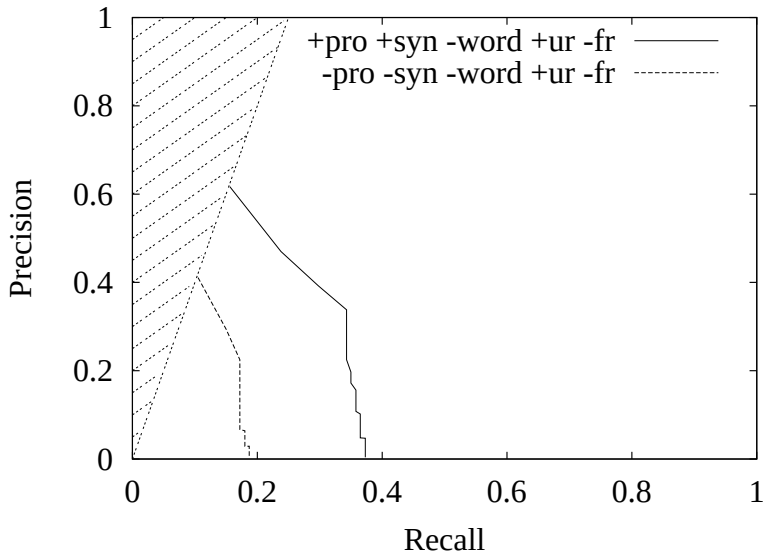

Figure 5-4: Precision-Recall Curve of SQUIRE Indexing Underspecified Relations.

5.4.2 Effects of the *ur* parameter

Indexing the underspecified relations in the corpus significantly increases SQUIRE's precision, as shown in Figure 5-4. Indexing just the underspecified relations, SQUIRE achieves a precision of 42%, and when augmented by the pronoun resolution and synonymy layers, it achieves a precision of 62%. This high precision reflects the fact that matching relations very reliably indicates that a sentence answers a question. In fact, when SQUIRE indexes only underspecified relations, it very rarely returns an incorrect answer; instead, it returns no answer at all.

However, that high precision comes at a cost: if a sentence's relations do not match a question's, then it will never be answered, even if they have almost identical word content. Thus, the precision drops off very quickly as recall increases, and SQUIRE never retrieved more than 22% of the correct answers when indexing just underspecified relations.

This cost can be overcome by indexing words in addition to relations. By indexing both words and underspecified relations, SQUIRE can achieve excellent performance, with a high initial precision that drops off slowly as recall increases. Figure 5-5 shows SQUIRE's performance when indexing both words and underspecified relations, and

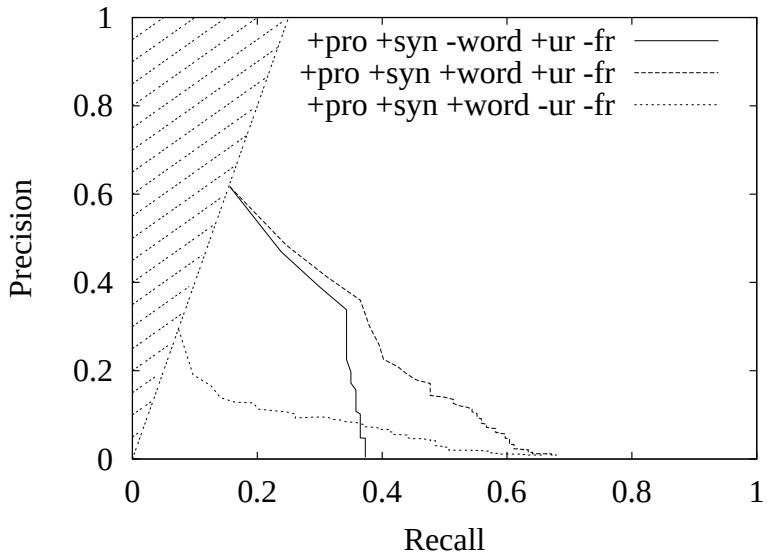

Figure 5-5: Precision-Recall Curve of SQUIRE Indexing Both Words and Underspecified Relations.

compares it to indexing either individually.

5.4.3 Effects of the *fr* parameter

Indexing full relations gives very similar behavior to indexing underspecified relations. Under almost every parameter setting, indexing full relations and indexing underspecified relations gives equivalent performance. In addition, indexing both types of relations also gives equivalent performance. Figure 5-6 compares the use of full relations to that of underspecified relations under a number of parameter settings.

The failure of full relations to improve the performance of SQUIRE probably stems from two sources. First, SQUIRE's algorithm for deriving semantic classes and relation types is still relatively primitive, and may not be reliable.

But there is a second, more fundamental, reason why full relations should not significantly improve SQUIRE's performance. The relation type, which is the primary additional information that distinguishes full relations from underspecified relations, is often unnecessary for the task of information retrieval. To see why, consider the sentence "elephants live in Africa." It is composed of two relations:

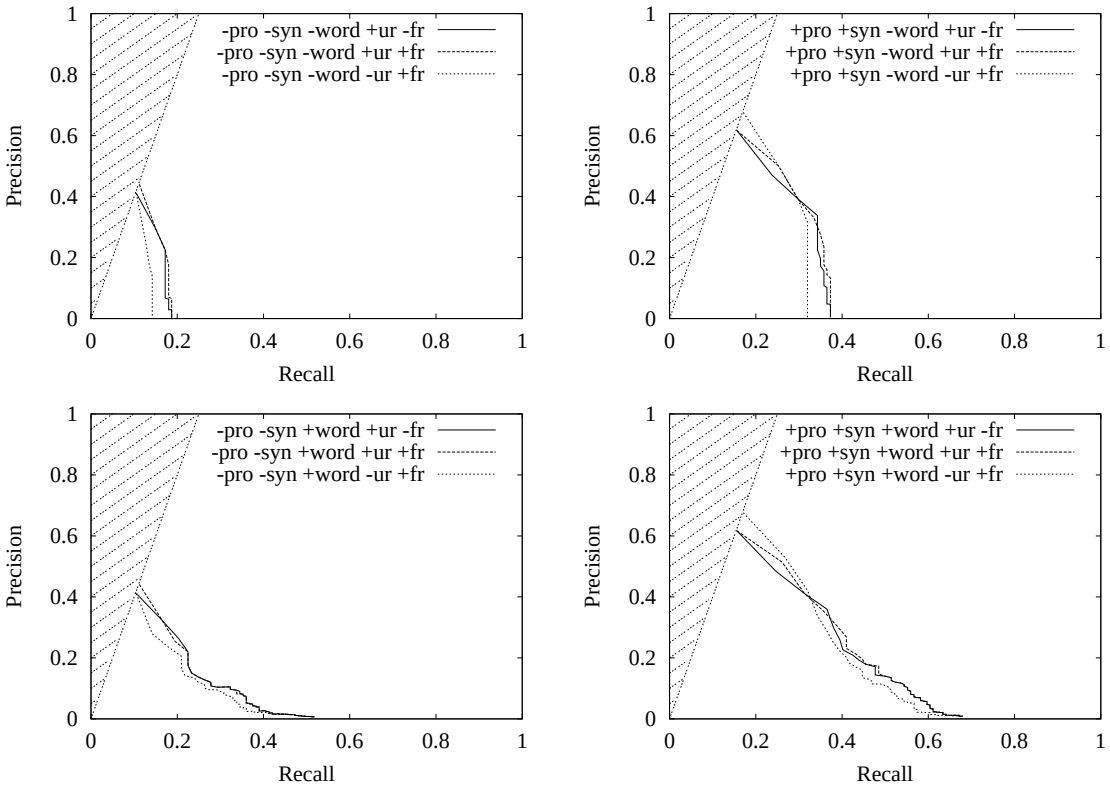

Figure 5-6: Comparison of Indexing Underspecified Relations to Indexing Tagged Relations

Semantic Head	Semantic Class	Adjunct	Relation Type
live	location verb	elephant	agent
live	location verb	Africa	location

Although the relation types are explicitly listed, they could also be derived from independent semantic constraints: only under exceptional circumstances can Africa be the agent of the verb "to live," or can elephant its location. Neither the corpus nor any question we encounter will ever contain these head/adjunct pairs with different relation types. Therefore, the relation types give us no additional information with which to decide whether two relations match.

Whenever the type of a relation can be reliably predicted from its head and its argument, finding that relation type will not improve IR performance. However, in sentences like "John likes Mary," where the arguments could easily be exchanged, full relations improve performance. Further research is necessary to determine how often the type of a relation can be predicted from its head and its argument. This research would give us a better indication of whether constructing full relations is worthwhile.

5.4.4 Effects of the *pro* parameter

The use of the pronoun resolution layer increases SQUIRE's precision and recall, regardless of the other parameter settings. The effect of this parser layer on SQUIRE's performance under a number of parameter settings is shown in Figure 5-7.

This uniform increase in performance is to be expected, since pronoun resolution replaces pronouns, which should never be used in matching, with useful values. This allows SQUIRE to match sentences that it wouldn't otherwise, whether it is indexing words, full relations, or underspecified relations.

As shown by Figure 5-7, the *pro* parameter has a stronger effect when full relations or underspecified relations are indexed than it does when only words are indexed. This may result from the decrease in weight when assigning a value to a pronoun. This decrease will have a stronger effect on word-based IR than it does on relation-based IR, since individual words are more common than individual relations. Thus, there will be more sentences with the same words but with higher weights that will give

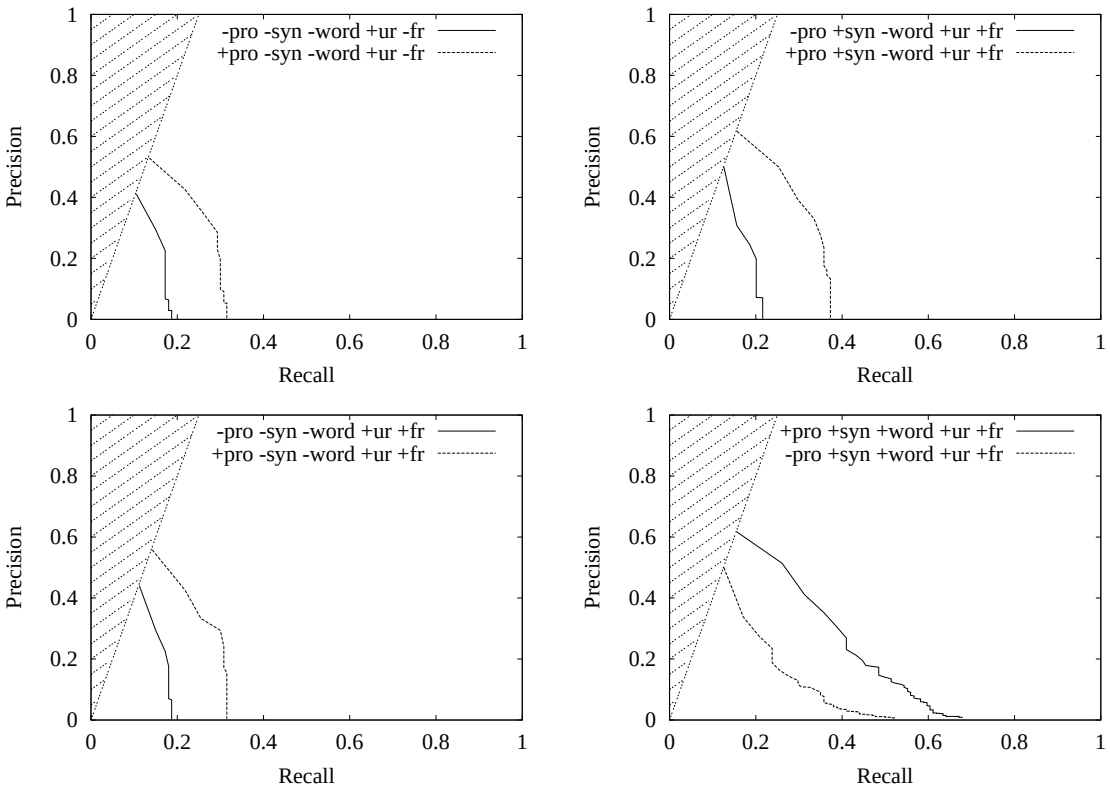

Figure 5-7: Precision-Recall Curves Illustrating the Effect of the Pronoun-resolution Parser Layer

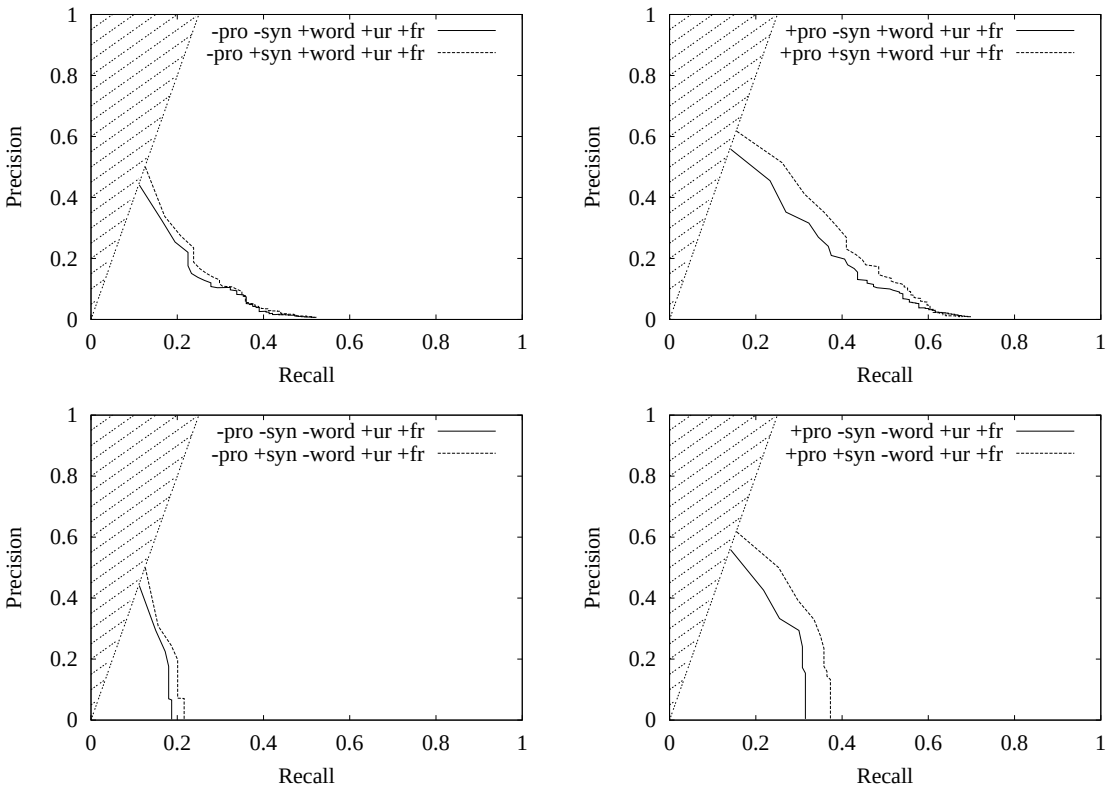

Figure 5-8: Precision-Recall Curves Illustrating the Effect of the Synonymy Layer when Relations are Indexed

better matches to the question.

5.4.5 Effects of the *syn* parameter

The use of the synonymy layer increases SQUIRE's precision and recall whenever full relations or underspecified relations are indexed, as shown in Figure 5-8. However, the effect is not not nearly as dramatic as that of the pronoun resolution layer. In fact, the synonymy layer actually slightly decreases SQUIRE's performance when only words were indexed, as shown in Figure 5-9.

The fact that the synonymy layer does not significantly improve performance is most likely a reflection of the fact that the algorithms used by the layer are not advanced enough to reliably find useful synonyms. The use of more advanced algorithms might allow the synonymy layer to yield more significant improvements to performance.

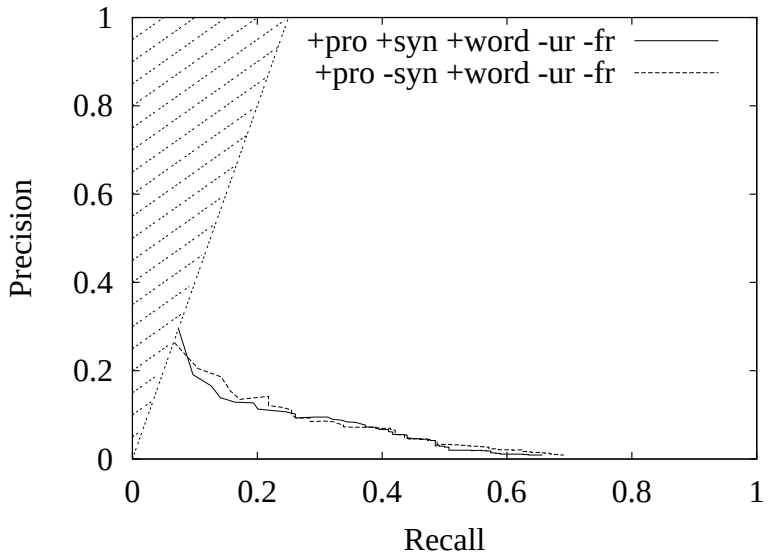

Figure 5-9: Precision-Recall Curves Illustrating the Effect of the Synonymy Layer when Only Words are Indexed

Another possible obstacle to the synonymy layer is over-generation of inappropriate synonyms. This over-generation leads SQUIRE to retrieve articles which are not actually relevant, simply because they contain the correct set of synonyms. However, this problem is reduced by by associating weights with the synonyms, which indicate how likely they are to correctly represent the original word.

5.5 Conclusions

Figure 5-10 compares the performance of SQUIRE, using all semantic IR techniques, to that of the control system. SQUIRE significantly out-performs the traditional IR system.

The three parameters which provide the greatest benefit to SQUIRE's performance are underspecified relation indexing, word indexing, and pronoun resolution. Underspecified relation indexing provides SQUIRE with high-precision answers, whenever a question's relations can be matched against a sentence's. Word indexing provides SQUIRE with a fall-back mechanism when all the sentences whose relations match have been exhausted. It thereby gives precision a gentle drop-off as recall increases.

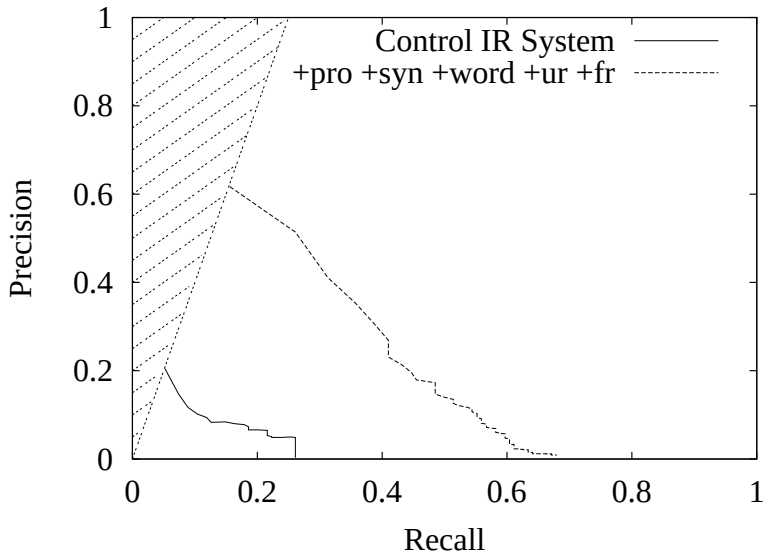

Figure 5-10: Comparison of the Performance of SQUIRE with All Features Enabled to the Performance of the Control IR System

Finally, pronoun resolution allows SQUIRE to match many sentences that do not explicitly contain the concepts they discuss.

In contrast, full relation indexing and synonymy detection do not seem to significantly improve SQUIRE's performance. While this lack of improvement may be caused by inadequate implementations, it may also stem in part from more fundamental limitations of these techniques.

Chapter 6

Future Directions

SQUIRE has shown that semantic-based techniques can far out-perform traditional techniques in the task of sentence level information retrieval. However, there is still a great deal of room for improved performance, and further research is necessary to explore the potential of semantic IR techniques. SQUIRE's layered parsing architecture provides a useful platform for this exploration.

6.1 Improving Existing Parser Layers

Several of the parser layers currently used by SQUIRE use primitive techniques, and do not always provide accurate results. These techniques were used so that the layers could be rapidly developed, and their potential could be tested. Now that these layers have demonstrated their usefulness, they should be updated to use more advanced techniques. In particular, the following improvements should be made to the algorithms used by the parser layers:

- The Stemmer is currently based on a large, pre-generated table of stems and words. As a result, it is unable to stem any novel words. The Stemmer should be re-implemented using rule-based or probabilistic techniques, or a combination of both.

- The Pronoun Resolver currently assumes that any pronoun refers to the subject

of the current sentence or one of the past two sentences. Also, it does not currently check for matching of such features as gender or number. Clearly, this is a vast oversimplification of pronoun behavior. The algorithms used by the Pronoun Resolver should be replaced with more accurate and linguistically-motivated techniques, such as those derived from Centering Theory [4].

- The Synonymy Layer currently contains only a handful of synonyms, and does not allow the user to specify synonyms with different semantic classes. The Synonymy Layer should be updated to use an existing source, such as WordNet, to derive its set of synonyms. It should also be updated to allow for synonyms with different semantic classes.

- The set of semantic classes employed by the Theta Assigner needs significant improvement before it can become a truly useful parser layer.

6.2 Creating New Parser Layers

New semantic processing techniques can be easily incorporated into SQUIRE's layered architecture, simply by constructing new parser layers. Once a new parser layer has been constructed, its effects on SQUIRE's performance can be easily tested. The interactions between the new layer and existing layers can also be explored.

For example, a parser layer could be constructed to use simple semantic implicatures. These semantic implicatures would be encoded in the form of rules. For example, consider the following rule:

(39) *?x* gave *?y* *?z* **implies** *?y* received *?z*.

This rule could be used by SQUIRE to decide that the sentence "Mary gave John an apple" answers the question "What did John receive?" The addition of such rules could be used to improve SQUIRE's performance within specific domains, while not hindering its performance outside those domains.

6.3 Fundamental Changes to SQUIRE

Although SQUIRE's architecture allows for rapid prototyping and testing of semantic IR techniques, there are a few fundamental problems with its current representations. Many of these problems arise from the fact that tree-structured representations are treated as atomic units. Thus, whenever a piece of a tree-structured representation is ambiguous, the whole structure must be duplicated once for every possible interpretation. This becomes especially problematic for sentences that are ambiguous in multiple locations. Consider the sentence:

(40) He wanted to give it to her, but he didn't know if she would want it.

This sentence contains six pronouns. Since the Pronoun Resolver currently assigns three possible values to every pronoun, it will convert this sentence into over seven hundred "possible" sentences. Each of these seven hundred sentences will have to be processed by the following parser layers. Clearly, this is a great source of inefficiency in SQUIRE.

However, the problem goes beyond a mere question of efficiency. In some cases, the inability to assign different weights to different parts of a tree causes more fundamental problems. Consider the sentence:

(41) John likes Mary and Ann.

Currently, this sentence is broken into two separate sentences:
(42) John likes Mary.
(43) John likes Ann.

However, it is unclear what weights these sentences should be given. Currently, SQUIRE assigns them the same weight as the sentence that they are derived from. However, that means that when they are broken into relations, the relation "John likes" and the words "John" and "like" will be given a higher weight than it would be given if SQUIRE were just indexing sentences (42) or (43). This is especially problematic for highly conjunctive sentences, such as:

(44) They feed on such animals as monkeys, antelope, muntjacs, jackals, peacocks, snakes, sheep, goats, and dogs.

In this sentence, the word "feed" is weighted about nine times as strongly as it

should be. As a result, this sentence was returned in response to almost every question about what any animal feeds on.

One solution would be to assign the derived sentences (42) and (43) half the weight of the original sentence, (41). This would give "John likes" the proper weight, but would result in "likes Mary" and "likes Ann" receiving weights that are too small.

The only way to circumvent this problem is to change SQUIRE's representations to allow individual parts of a tree-structured representation to have multiple possible values, each with its own weight.

Bibliography

[1] K. Andrews. *The Development of a Fast Conflation Algorithm for English.* PhD thesis, University of Cambridge, 1971.

[2] Avi Arampatzis, Th.P. van der Weide, C.H.A. Koster, and P. van Bommel. An evaluation of linguistically-motivated indexing schemes. Technical Report CSI-R9927, University of Nijmegen, The Netherlands, December 1999.

[3] A. Bookstein. When the most "pertinent" document should not be retrieved – an analysis of the swets model. *Information Processing and Management*, 13:377–383, 1977.

[4] S. E. Brennan, M. W. Friedman, and C. J. Pollard. A centering approach to pronouns. *Proceedings of the 25th Conference of the ACL*, 1987.

[5] Eric Brill. A simple rule-based part-of-speech tagger. In *Proceedings of the Third Conference of Applied Natural Language Processing*, pages 152–155, Trento, Italy, 1992.

[6] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference (WWW7)*, Brisbane, Australia, 1998.

[7] B. C. Brookes. The measure of information retrieval effectiveness proposed by swets. *Journal of Documentation*, 24:41–54, 1968.

[8] C. Buckley, G. Salton, and J. Allan. The effect of adding relevance information in a relevance feedback environment. In *Proceedings of the 17th Annual Inter-*

national ACM SIGIR Converence on Research and Development in Information Retrieval, pages 292–300, Dublin, Ireland, 1994.

[9] Google homepage. `www.google.com`.

[10] Barbara J Grosz and C. L. Sidner. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[11] Liliane Haegeman. *Government and Binding Theory*. Blackwell Publishers Inc., Cambridge, MA, 2 edition, 1994.

[12] Benjamin Han and Rohini K. Srihari. Text retrieval using object vectors. Technical report, State University of New York at Buffalo, 1999.

[13] Irene Heim and Angelika Kratzer. *Semantics in Generative Grammar*. Blackwell Publishers Inc., Camridge, MA, 1998.

[14] K. S. Jones. What is the role of nlp in text retrieval. In T. Strzalkowski, editor, *Natural Language Information Retrieval*. Kluwer Academic, 1999.

[15] Boris Katz. Using english for indexing and retrieving. In *Artificially Intelligence at MIT: Expanding Frontiers*, volume 1. MIT Press, 1990.

[16] Boris Katz and Beth Levin. Exploiting lexical regularities in designing natural language systems. In *Proceedings of the 12^{th} International Conference on Computational Linguistics, COLING '88*, Budapest, Hungary, 1988.

[17] Angelika Kratzer. Severing the external argument from its verb. In J. Rooryck and L. Zaring, editors, *Phrase Structure and the Lexicon*. Kluwer Academic Publishers, 1996.

[18] Beth Levin. *English Verb Classes and Alternations: A Preliminary Investigation*. The University of Chicago Press, Chicago, IL, 1993.

[19] Jimmy Lin. The strata system: Natural language information access. Master's of Engineering Thesis Proposal, 1999.

[20] William O'Grady, Michael Dobrovolsky, and Mark Aranoff. *Contempory Linguistics: An Introduction.* St. Martin's Press, Inc., New York, NY, 1991.

[21] T. Parsons. *Events in the Semantics of English*, chapter 5. MIT Press, Cambridge, MA, 1990.

[22] S. E. Robertson. The parametric description of retrieval tests, part 2: 'overall measures'. *Journal of Documentation*, 25:93–107, 1969.

[23] G. Salton, editor. *The Smart Retrieval System: Experiments in Automatic Document Processing.* Prentice-Hall, 1971.

[24] A. F. Smeaton. Using nlp or nlp resources for information retrieval tasks. In T. Strzalkowski, editor, *Natural Language Information Retrieval.* Kluwer Academic, 1999.

[25] J. A. Swets. Information retrieval systems. *Science*, 141:245–250, 1963.

[26] Vivien C. Tartter. *Language and its Normal Processing.* Sage Publications, Inc., London, UK, 1998.

[27] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, UK, 1979.

[28] Ellen M. Voorhees. Using wordnet for text retrieval. In C. Fellbaum, editor, *Wordnet: An Electronic Lexical Database*, chapter 12. MIT Press, Cambridge, MA, 1998.

[29] Worldbook encyclopedia homepage. `www.worldbook.com`.