Enhancing Information Retrieval using Syntactic Relations

by

Igor Kaplansky

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

February 2001

© Igor Kaplansky, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute
publicly paper and electronic copies of this thesis document in whole or in part.

Author .................................................................
Department of Electrical Engineering and Computer Science

February 8, 2001

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

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

BARKER

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 11 2001
Enhancing Information Retrieval using Syntactic Relations

by

Igor Kaplansky

Submitted to the Department of Electrical Engineering and Computer Science on February 8, 2001, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

As the amount of information available through the web increases rapidly, it becomes crucial to develop information retrieval tools that would be able to access the right information as fast as possible. The majority of question answering systems available today use the traditional bag of words approach. Within this approach documents are treated merely as a collections of words. All other information contained within a document is disregarded. While being fairly effective, this approach is far from satisfactory because the user needs to browse through a pile of irrelevant articles in order to find the necessary information. In this thesis I have implemented an information retrieval system that combines keyword based indexing with indexing syntactic relations. Preliminary testing has showed that indexing syntactic relations alone improves precision while degrading the recall. However, when both methods are combined, the resulting system performs better than the keyword based engine alone.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist
Acknowledgments

I would like to thank Boris Katz, my academic and thesis advisor, for providing valuable ideas as well as helping me through the 4 years I’ve spent at MIT. I also would like to thank Ali Ibrahim, Rebecca Schulman and Jimmy Lin for working with me and sharing their experience in the area of natural language understanding.
Contents

1 Introduction 6

2 Research 7
   2.1 Information Retrieval Techniques 7
       2.1.1 Limitations of Traditional IR techniques 7
   2.2 START Natural Language System 8
   2.3 Directory based indexing 10
   2.4 NLP techniques in Information Retrieval 10

3 SIREN Description 11
   3.1 PS Tagger 12
   3.2 Indexer 13
   3.3 Query Generator 14
   3.4 Query Evaluator 15
   3.5 Scoring Mechanism 16
   3.6 Minipar Relations Parser 17
       3.6.1 MINIPAR Grammatical Relationships 17

4 Algorithmic Details 19
   4.1 Indexing 19
   4.2 Creating temporary files 19
   4.3 Creating backward index files 20
   4.4 List Intersection Algorithm 21
4.5 Relation Indexing .................................................. 22

5 SIREN Performance .............................................. 24
   5.1 Testing Method .................................................. 24
      5.1.1 Test Corpus .............................................. 25
      5.1.2 Text Questions .......................................... 25
   5.2 Results and Statistics ....................................... 25
   5.3 Conclusion ..................................................... 26

6 Future work ....................................................... 28
   6.1 Changes to SIREN ............................................. 28
   6.2 Synonyms, Hyponyms ......................................... 28
   6.3 Improving Query Generator ................................ 29
   6.4 Improving Relation based queries .......................... 29
   6.5 Pronoun Resolution .......................................... 30

A Setting up a New Corpus ....................................... 31
Chapter 1

Introduction

Being able to ask a computer anything in plain English and receive a reply indistinguishable from that by the Star Trek enterprise computer is many people's dream. Meanwhile, search engines are our primary tools for searching through large, rarely organized archives of information, such as the World Wide Web, newspaper archives, and electronic books. Unfortunately, search engine technology is currently limited in most cases to keyword searches which makes it a very weak match for its Starship Enterprise colleague. One of the problems with keyword-based search engines, extensively described by many researchers, is that they treat documents as a bag of words and disregard all syntactic and semantic information contained within documents. For example if you type into your favorite browser even a simple question like "what do cats eat?" you will receive thousands of articles and maybe only a few of them (buried deeply inside) might actually answer your question. Syntactic relations based indexing is an attempt to improve on the keyword based system by indexing not only individual keywords, but relations such as possessives, subject-verb-object, modifiers, prepositional clauses among others. In this thesis I built a system SIREN which makes use of syntactic relations contained in the text by indexing them along with words. Preliminary experiments have shown that indexing syntactic relations improve precision at the expense of recall. However combining both methods yields a system that outperforms a standard keyword based one.
Chapter 2

Research

2.1 Information Retrieval Techniques

Automatically searching through a collection of text with the goal of retrieving some specific information is known as Information Retrieval. Traditional IR systems break down the corpus into a collection of articles. Each article discusses some particular topic. All the words contained in the articles are usually indexed such that when a question is asked it would be possible to return all articles that contain the same words as the question. Such an approach is called “bag of words” because all words are treated equally and none of the syntactic or semantic relations between the words are retained. An example of such system are the majority of search engines available on the world wide web, as well as SMART IR system [10].

2.1.1 Limitations of Traditional IR techniques

When a document is treated as a bag of words, all the relations between the words are thrown away. This makes the following sentences indistinguishable for a traditional IR system:

   The man is biting the dog. The dog is biting the man.

   Both sentences contain the same words, so if we disregard syntactic relations contained in the sentences, asking “Did the dog bite the man?” or “Did the man bite the dog?” would match both of them.
Another major limitation of a traditional IR system stems from the fact that the exact wording of a question rarely matches the way that information appears in the corpus. Therefore the traditional approach will fail to answer all questions whose wording is different from the one in the corpus. Here are some examples of such possible questions and answers:

- who was the first man to walk on the moon?

  Neil Alden Armstrong, a United States astronaut, was the first person to set foot on the moon.

- When did Bill Clinton become a president?

  William Jefferson Clinton was elected president in 1992

Finally, major concepts, ideas, people’s names and so on are named usually in the beginning of an article. The rest of the times they are referred to by pronouns or other mechanisms like paraphrasing and synonymy. For example:

- how is the magnetosphere shaped?

  It is shaped like a teardrop, with the point extending away from the sun.

In order to solve these problems it’s necessary to extract semantic and syntactic information contained in the text and match it against the information contained in the question. The system that I present in this thesis is a step in that direction.

2.2 START Natural Language System

Indexing individual words and syntactic/semantic relations is not the only alternative on the way to a natural language based question answering system. A different approach involves developing a full natural language based parser. But even if you have such a parser, it is not at all clear how to represent and use the information provided by it. In addition, there are a lot of difficult problems that further complicate the issue, such as intersentential reference, paraphrasing, common sense implications and so on. Each of the mentioned problems is a separate research field on its own.
A step in the direction to a full natural language parser is represented by the START system developed by Boris Katz, which processes English sentences and stores them in its database in the form of ternary expressions. A ternary expression (T-expression) is a subject-relation-object structure where both the subject or the object can be T-expressions themselves. For example, the sentence "John eats the big orange" would be converted into the following triples: [John eat orange-1] and [orange-1 is big].

To answer any question posed in English, the question is first transformed into its declarative form and then into a corresponding T-expression. The resulting T-expression is then matched against START's knowledge base. If a match is found, the corresponding T-expression from the knowledge base is then converted, using a language generator module, into a natural language sentence. Going back to the example, when START is given a query like "What did John eat?", the query will be first converted into a triple "John eats what" and then matched again against the database of indexed relations.

One of the problems that START had to face was finding a way to equate sentences with different surface structures but identical or very similar underlying (deep) structures ([1],[2]). By means of a special mechanism (S-rules), START can ensure that various syntactic realizations of the same meaning are related ([2]).

START is quite successful at answering questions posed in English. However, the system has become complex over time and faces some limitations. For example, indexing large amounts of information would be inefficient using the current indexing scheme.

Two additional mechanisms, schemata and annotations, have improved START’s ability to understand a wide range of questions. They allow additional knowledge to be added to START’s database in a relatively straightforward manner [13]. A downside of this method is that each data source needs to be added by hand by writing additional annotations and schemata that require human intervention. Parameterization alleviates the problem by allowing a single annotation to apply across a class of knowledge, but there is still a nontrivial manual labor involved which ideally would be avoided. Therefore one of the potential applications for my system is to enhance START’s ability to index and retrieve large amounts of information.
2.3 Directory based indexing

Directory-based indexing (e.g. Yahoo) is an attempt to enhance a regular keyword-based search with a collection of pointers to various websites which is hierarchically organized according to some semantic categories like entertainment, business etc. While it is an improvement over a pure search (improved precision), this approach does not really address the problem of natural language understanding but rather aids the user to navigate through the slightly better organized mess of information. In addition, the hierarchy needs to be manually maintained and its organization is subjective (different people assign categories to the websites) and therefore bound to be either incomplete, incorrect or outdated as time goes by.

2.4 NLP techniques in Information Retrieval

In a standard keyword-based system, documents are reduced to a "bag of words" and all the semantic and syntactic information contained in the text is ignored [8]. This is the main reason why searching through the Web can be a very frustrating experience. In such a framework, documents are viewed as a high dimensional vector whose components are individual words weighted according to some scheme.

The task of determining the relevance of the document to the query reduces to calculating the distance between the document’s and the query’s vectors. It would seem that taking advantage of the semantic and syntactic information contained within a document should improve the situation. Even though some research done in the last couple of decades indicates inconsistent results [3,9], the NLP-based text retrieval systems do show an improvement over a conventional "bag of words" model. The system implemented by B. Han and R. Srihari [4] represents documents by identifying syntactic groups and semantic relations between them. Semantic relations between objects are based on the word forms and ordering of the phrase constituents. A similarity score for a pair of objects is computed by traversing semantic links in WordNet. Preliminary testing was done on a collection of 500 articles but no conclusive results were obtained.
Chapter 3

SIREN Description

SIREN is a system that combines a standard keyword based approach to answering a query with a relations based one. The input to the system can be either a sentence posed in English or just a series of individual keywords.

When operating in a relations mode, SIREN uses MINIPAR parser (which will be described later) to extract syntactic relations contained in the input sentence. Then it attempts to match extracted relations with the ones contained in its relation database. If matching succeeds, it returns the collection of sentences or articles that contained the matched relations.

In the keyword based mode, Query Generator processes the sentence or a series of keywords to construct a query object which is then used to query the database of words. If matching succeeds, each individual match is scored. Finally, all matches are sorted and returned to the user.

SIREN is composed of the following parts:

- **PS Tagger** - a wrapper for the Brill’s part of speech tagger (will be described later) which puts the corpus text in a format suitable for indexing.

- **Indexer** - a module that indexes individual keywords and syntactic relations as well as other related information (ie. sentence and article numbers, part of speech)

- **Query Generator** - generates a query object from an input sentence or keywords.
3.1 PS Tagger

Part of speech information is used in SIREN during scoring to give a higher score to the words that have identical part of speech in the question and in the corpus text. To tag each word appearing in corpus with its part of speech SIREN uses a statistical part of speech tagger written by Brill [12].

The input to the SIREN’s PS Tagger module is a file containing the text to be indexed. The format of the input file is the following: articles (or any other unit of text) separated by a new line. Tagger then creates a file where each sentence occurs on a new line, each text unit (article or paragraph) is separated by a new line and each word is tagged with its part of speech (Brill’s POS tagger) and is separated by a tab. Sample output paragraph of the Tagger would look like this:

\[
< NN\text{str} = "Afghanistan" > < NN\text{str} = "history" > <: str = ":" > ...
\]

In the above sentence, “NNP” tag stands for a common noun, while “NN” tag stands for a common noun.

I have also added a special META tag that can be used to associate some extra information known about an article which is not necessarily part of the article. This information can be later used in the scoring routine. A sample META tag for Factbook99 corpus looks like this:

\[
< META country = Afghanistan&attribute = AGE – HTML >
\]

This particular tag has two pieces of information: the country name and an attribute name that can be later used to extract the article from Omnibase (START’s database).

Each article can have any number of META tags associated with it. It is possible to extend the framework such that META tags are associated also with individual sentences.
3.2 Indexer

The Indexer module is built in C. Using a commercial database would be more robust but somewhat of an overkill since most of its functionality wouldn’t be used. Due to its simple design the module efficiently indexes syntactic relations extracted by MINIPAR parser as well as keywords.

- **Indexing individual words**

  Many words that appear in texts have various affixes like “-s”, “ed”, “ing” which contribute very little to the semantics of the words. If information retrieval systems treated all of these affixes as part of the word, only words that contain the same affixes as those in the question would be returned. For example, the system would not return the sentence “Cat drinks milk” in response to the question “What do cats drink?” because the words “cat” and “drink” have different affixes in the answer and question sentences. For this reason, most information retrieval system strip the words of their affixes and index only the root form of the word.

  SIREN indexes words as they occur in a corpus, their lowercased versions as well as their stems. For example, if the word “Russian” occurred in the corpus, then all following words would be indexed: “Russian”, “russian”, “russia”. The rational for indexing an upper and lower case versions of a word is that this could be used later in the scoring mechanism. The scoring module could assign a higher value to the words that appear both in the query and in the corpus identically (ie. capitalized, all capital letters and so on)

  SIREN also indexes article and sentence numbers where individual words occur, their part of speech and information content. Article and sentence numbers are used later in retrieving the results of the query, while part of speech and information content are used during the scoring phase.

- **Indexing syntactic relations**

  SIREN indexes the relations extracted by the MINIPAR parser. A sample relations is a triple that consists of the relation head, type and modifier. For example, given
a sentence "John loves Mary", some of the extracted relations would be "love subj John" and "love obj Mary". SIREN indexes these relations in the following way: all three components of a "triple" are concatenated together to create a string which is then hashed. For example, for the two relations above SIREN would create two corresponding strings "love_subj_John" and "love_obj_Mary" which would be hashed using the same scheme as the one used for keyword indexing.

Indexing each component individually would allow less constrained relations queries. That would allow to retrieve all relations that only match the head and/or the modifier of a relation. This is not done currently.

### 3.3 Query Generator

Query Generator is a module that transforms the incoming question into the format suitable for querying the database when SIREN is operating in a keyword based approach. It strips off functional words such as prepositions, pronouns and adverbs which do not contribute much to the meaning of the text and leaves out content words that are actually useful for a keyword query.

The query which is constructed by the QG can be of two types: same sentence (SS) and same article (SA). The type of the query just implies that the results are either a collection of articles containing the query keywords or a collection of sentences.

For example a question like: "When did Clinton become a president?" would result in the following query being constructed:

(SS clinton///ps=NNP///original=Clinton preside///ps=NN///original=president become///ps=VB///original=become)

This query contains 3 keywords: clinton, preside and become which are the root forms of the original words. Part of speech information is also present in the query.

A query object could also be nested, ie. some of its arguments can be queries themselves. This is an example of a nested query:

*Question:* Who was the first american president to be impeached?
Query:

(SA

(SS *first///ps=JJ///original=first
american///ps=NN///original=american
preside///ps=NN///original=president)
impeach///ps=VBN///original=impeached)

This query contains 2 arguments. The first one is a query which contains 3 keyword arguments while the second is a keyword (impeach). The system constructed this query using the following simple algorithm: first it extracted contiguous noun phrases from the sentence. For each of the noun phrases a SS type query was constructed. Then the system combined SS queries into one SA type query adding in contiguous sequence of verbs contained in the question.

The Query generator can also mark an individual keyword argument with an asterisk which makes the evaluation engine treat it as optional (useful in the case of some adjectives). In this case the query will not fail if the optional word is not found.

In the future the query generator can also be used to determine the type of a question (what type/kind of, who, when, where) in order to determine what kind of query should be constructed to give the best answer. For example, given a question "What type of birds migrate to south?", the Query Generator will figure out that it should use a migrate-subj-obj relation index and that "bird" is a generalization for a family of animals so the object position of the matched relation should contain some animal whose hyponyme is "bird".

3.4 Query Evaluator

Query Evaluator retrieves matched articles or sentences for each query argument, finds an intersection and then scores the final matches according to the chosen scoring scheme. The current implementation looks up the query keyword in the word hash table to get a "word code" for the word, which is how it is stored in the database. The word code determines
the position in the backward index reference file where 2 numbers are stored. The numbers represent the range of matches for that word. In order to retrieve the actual matches, article index or sentence index files used.

3.5 Scoring Mechanism

In the keyword mode the system doesn’t have any way of deciding which matches are better or worse based on the syntactic or semantic information. It only knows what kind of words appeared in the query and which words in the corpus matched them. Nevertheless, simple heuristics like word proximity allow to significantly improve the precision of the system.

A scoring scheme which is currently implemented takes into account an individual keyword position and its proximity to the beginning of a sentence, paragraph or an article. The closer the word to the beginning of a sentence the greater the chance that it might be relevant. Matches in which many keywords are situated closer together receive a higher score as well.

Scoring module also looks at the part of speech tags of the words in the query and in the corpus. Words that have identical POS tags in both places receive a higher score.

One additional piece of information available for scoring is the information value of the words. Information value of the word is a measure of how much information a word contains or in other words how common or uncommon a word is. For example, the word “the” is extremely common, therefore its information value would be very low. Words with a greater information value receive a higher score.

Since matching is done on an individual keyword level, it’s possible to get many matches for the same word that belong in the same article. In this case the score for all such matches is also increased.

In the future I am planning to implement some heuristics based mechanism that would give different weights to relations. One heuristic could be to rank all relations according to some importance measure. For example, “subject verb object” relations would rank higher than “modifier” type relations. The total number of matched relations could also indicate how relevant is a given match. The relative importance of relations can depend on the
type of question asked. For example, in a question “Who is the president of the Russian
government?” the relation “government mod Russian” should probably be given the most
weight. On the other hand, in the question “What do wild hogs eat? “ the relation “hog eat
*” should be given more weight than “hogs mod wild”.

3.6 Minipar Relations Parser

The role of the parser is to convert the input sentence strings into relations. I used MINIPAR
parser designed and built by Dekang Lin. The parser first builds syntactic trees that represent
the underlying sentences. Then by traversing the trees it creates relations each of which
consist of 3 parts: head, relation type and modifier. The following section describes all the
relations that SIREN indexes:

3.6.1 MINIPAR Grammatical Relationships

The following list contains the relations extracted by MINIPAR and which are indexed by
SIREN.

appo "ACME president, P. W. Buckman, committed suicide."
aux "I think you should resign"
be "He is sleeping."
c "I think that John loves Mary"
det "I poked the brain with a screwdriver."
gen "Jane’s uncle is an idiot." (possessive)
have "They all have disappeared by the time the food came."
inv-aux "Will you stop it?"
inv-be "Is she sleeping?"
inv-have" Have you slept”
mod "A green apple fell on the floor."
pcomp-n “Garry hit John with a stone"
wha, whn, whp "What have you done?"

obj "Cats eat mice"

subj "dogs chase cats"

Current scheme for matching relations is rather simple. Relations of type wha, whn,whp are ignored for lack of an appropriate mechanism that would use these relations to determine what exactly is being asked and what relations should be used in matching to get the best answer. The rest of the extracted relations is matched individually against the relation database. At the end all results are intersected to find articles or sentences that contain all the matched relations.
Chapter 4

Algorithmic Details

4.1 Indexing

This chapter will explain how a corpus is indexed so that queries may be immediately processed at run time.

The purpose of the indexing stage is to create a backward index. The index lists all the locations where each word occurs in the corpus. When a query is read, the word being asked about is looked up in a hash table. The word code (hashtable slot) determines the unique position in the backwards index reference file, where two integers are stored: the start and the end places of the locations for that word. To actually retrieve the matches for the word, one would need to read in (stop-start) integers from a corresponding article or sentence index file.

4.2 Creating temporary files

A corpus file is first converted to a format suitable for indexing. The corpus size can be arbitrarily big since all of the information is stored on disk. The temporary file is created in the following way: each word in a document is hashed resulting in some unique word code for the word. A temporary array is maintained to keep track of the last position (article, sentence) the current word was seen at. The array is initialized to -1 implying that none of
the words have been seen so far. To give an example, take a word "table". Its word code is first retrieved through hashing: 1234. Then in the position 1234 of last position array, the number 0 would be stored. Before this happens, the system stores -1 at the first position in the temp file, meaning this is a word that hasn’t been seen yet. Thus when the eighteenth word is again "table", the fifth integer in the temporary file will be 0, and the last position array slot corresponding to the word "the" will have the number 5.

4.3 Creating backward index files

Given the temporary file, we are ready to construct a backwards index file. Given the hash table, position array, and temporary file, both a backwards index, and a reference file for the backwards index are created using the following algorithm:

- Look at the the first slot in the hash table.
- If there is no word in it, write the same start and stop positions to the backwards index reference file.
- If there is a word, look up its hash code in the position array (the array would contain the last position at which that word was seen in the corpus), for example 4444.
- The position is written to the word index index file.
- To find the remaining positions where the word occurred, use the temporary file in the following way: Seek to the 4444th position in the file; the number at that spot represents the position previous to 4444 at which the same word was found.

Iterate this process until -1 is encountered at which point all positions for the word has been written out.

- Simultaneously keep track of "start" and "stop" positions that will be written to the back index reference file. Two counters are used. One is the word count, which keeps track of the number of words total that have been written, and the other is the "start" variable, which only changes when a new slot in the hash table is being considered,
at which time it is updated to the current word count. When a -1 is encountered, the
current start and stop are written to the backwards index reference file.

The rest of the words in the hashtable are processed similarly. When all words are
processed, the temporary file is erased.

4.4 List Intersection Algorithm

Intersection algorithm iterates down all the lists of integers in list simultaneously, looking
for places where all the arrays contain words that are close to each other. To support this
iteration, it creates an array called "pos" which is initialized to all zeros. As the iteration
continues, eventually all of the pos elements reach the corresponding int *length values.

Our algorithm is a combination of the two standard algorithms for intersecting sorted
lists with running times O(n) and O(lgn). One is based on a linear scan through the lists
keeping track of the common elements (hence O(n)* number of lists). The other one uses
binary search. Here is an informal description of the second algorithm:

- Choose your pivot list (ie. a list with fewest elements).
- Pick an element from your pivot list
- Use binary search to search for this element in the rest of the lists.
- Repeat steps 2 and 3 for the remaining elements of the pivot list.

Let $M = \text{length of the pivot list}$, $N = \text{length of the current list we are intersecting with}$. Then the search method is chosen in the following way:

$$if(M < (N/(my\text{Log}(N) - 1)))$$

use BINARY search

else

use LINEAR search
4.5 Relation Indexing

Indexing of relations is done in a way that is very similar to the keyword indexing procedure described above. Only now all three components of the relation are concatenated together and hashed in the relation hashtable. Besides relation positions I also keep track of relation article and sentence numbers. Hence I use 3 arrays as opposed to 1 when creating a temporary file. That allows me to create position, article and sentence index file for all of the relations. Relation positions are necessary if components of a relation are indexed and matched separately. Relation positions in this case would be necessary to enforce that parts of the relations matched actually belong to the same relation.

For example, right now a sentence “John loves Mary” contains two MINIPAR relations which I index:

“love subj John” and “love obj Mary”

Imagine that the following sentence is contained in the text “John loves Mary and Peter loves Sue” which implies that I would contain the following relations in the database:

“love subj John”, “love obj Mary”, “love subj Peter”, “love obj Sue”

When a question “does John love Sue?” is asked, the following relations are extracted:

“love subj John”, “love obj Sue”

If only article or sentence numbers are used to pinpoint the relation position then the answer to the question would have to be incorrectly “yes” because both relations extracted from the question are contained in the database. To solve this problem we would need to know when subject and object relations are in fact part of the same relation “SUBJ love OBJ”. Both relations would be assigned some unique number which would be later used in an intersection algorithm.

An alternative approach would be to identify which relations can be collapsed into one and index them in such a way. So instead of indexing “love subj John” and “love obj Mary” separately, we can index just one relation: “John love Mary”. In this case we wouldn’t make
a mistake as in the example above. However, we would run into trouble trying to answer questions like “who does John love?”. This happens because of the way SIREN indexes relations (which is described above) but it’s also very easy to fix by indexing each part of the relation separately.

Another alternative for matching “john_love_*” would be to add in two hashes. One of them would be keyed on subject_relation string, while the other hash would be keyed on “object_relation” string.
Chapter 5

SIREN Performance

The performance of SIREN was measured using Worldbook Encyclopedia and a set of about 40 questions about animals.

5.1 Testing Method

There are two standard metrics used to evaluate the performance of an information retrieval system:

Precision - the percentage of responses that answer the question

Recall - the percentage of the correct answers that are retrieved

To test SIREN, each of the 40 test questions were answered using three methods: standard IR with scoring, relations based, and a method that combined both mechanisms. The combined method at this stage is pretty straightforward: try to get a match using relation matching. If no results returned, run a standard keyword query with scoring. The answers to the questions were found by looking through all the articles in the encyclopedia that would be relevant to the question. I used precision-recall measurement method similar to the one used by Edward Loper in his SQUIRE system [11]. Namely, precision and recall were calculated the following way: for every n, 0 < n < 100, the precision and recall values were calculated as follows:
\[ p_n = \frac{A \cap B_n}{B_n} \quad r_n = \frac{A \cap B_n}{A} \]

A is the set of sentences that correctly answer the question, \( B_n \) is the first \( n \) sentences returned by SIREN in response to the question.

### 5.1.1 Test Corpus

For testing I used worldbook encyclopedia, a very structured and moderate in size corpus. Worldbook contains a wealth of information on various subjects which makes it a convenient corpus for experimental and testing purposes.

**Worldbook size:** 20337 articles, 590446 sentences, 50 MBytes.

### 5.1.2 Text Questions

The question set is a collection of about 40 questions about animals. Some sample questions would include:

*What do cats eat?*

*Do cats eat mice?*

*Who feeds on spiders?*

The chosen questions are simple enough to demonstrate a potential benefit of using syntactic relations for matching queries as opposed to keywords. In the future some experiments need to be run in order to determine which relations should be trusted more and how their results should be combined in order to answer more complicated questions.

### 5.2 Results and Statistics

The experiments have shown that the precision of SIREN in relation only mode was improved by approximately 15% while its recall degraded by 40% in comparison to a standard IR system. The results can be explained by the fact that a very specific set of questions was used for testing. Namely, questions of the type “What do cats eat?” are bound to be picked
up by the relation engine which would explain the improvement in the precision over a traditional IR system. The degradation of recall is explained by inaccuracies of the relations parser, which can miss relations due to the algorithms being used, inconsistencies in the text layout and simply the difference between relations extracted from the question and those appearing in the corpus.

When both systems were combined, the precision of SIREN was also improved by 20% while its recall suffered by 25% Overall, using relations as well as keywords proved to be a promising direction to follow even though additional experiments need to be carried out to make a more conclusive judgement.

SIREN is also a fairly efficient system as the following statistics show. These are some of the average times for indexing, relation extraction and query answering. The measurements were made on Anatolia, dual processor, 2G RAM Sun machine.

Keyword Indexing times: **Infoplease:** 20 sec. **Worldbook:** 35.3 minutes

Relation Indexing times: **Worldbook:** 6 min. **Infoplease:** 20 sec.

Extracting Relations time: **Worldbook:** 7 hours.

Corpus Initialization times: **Worldbook:** 35 s **Infoplease:** 15 sec.

Average keyword query response time: 0.2 sec

Average relation query response time: 0.1 sec.

### 5.3 Conclusion

SIREN has showed that combining a keyword approach with syntactic relations results in a system that has a better precision than a standard keyword based system. However, it also demonstrated a need for a more sophisticated algorithm for dealing with syntactic relations, otherwise the recall of the system can suffer significantly. Indexing relations improved precision by 15% while hurt recall by about 30% on a set of relatively straightforward questions. Even though further experiments are necessary to make a better conclusion as to how
syntactic relations should be used when answering a query; the current state of the system presents a good platform for future explorations.
Chapter 6

Future work

6.1 Changes to SIREN

A major improvement for the system would be rethinking the datatypes that are needed, and their interface. This would significantly ease the process of updating and extending the system with new features.

6.2 Synonyms, Hyponyms

One of the major problems of standard keyword based systems is that questions are almost never worded the same way as the answers. There isn't a phrase in the whole Worldbook Encyclopedia that explicitly states “cats eat mice”. There are however the following sentences:

*Cats hunt mice.*

*Domesticated wildcats killed mice.*

In order to alleviate this drawback at least partially, an indexer should also know about at least a few most relevant synonyms and hyponyms of words. I suspect that doing so would boost the performance of SIREN by a large factor. In order to do that WordNet or some other source could be used [6]. For example, when indexing a relation “John built
house”, the Wordnet might be used to extract the most likely synonym of the verb “built”: “constructed” as well as the most closely related hypernym of the object “house”: “home”. Then one could add two more relations “John constructed house” and “John build home”. Indexing these new relations along with the original one would increase the chance of a better recall for the query.

Indexing all possible synonyms of relations would be impractical since it would lead to an exponential increase in space required for indexing and would degrade precision. Ideally, it would be useful to have a separate layer that disambiguates word meanings in various contexts. Before we have such a module, we could compromise by indexing the most likely synonyms of the verbs that appear in the relations, and a few most common hyponyms of the nouns.

6.3 Improving Query Generator

Query Generator could be used to determine which database and relation indices should be used in the retrieval stage based on the syntactic/semantic information present in the question. This could be useful if the number of indices is huge and can not be contained in memory at one time. Query Generator also needs a better degradation mechanism. In the event when a query returns zero matches, it needs to be able to relax the query without oversimplifying it.

6.4 Improving Relation based queries

Currently relations queries are not scored at all. In the future it would be useful to run some experiments to determine what relations are more important and therefore should be given more weight.

When indexing relations as in the case of keywords it might be useful to index not only the way relation words appear but also the root forms of the words in the relation. For example:

Phrase: working people

29
Such expansion might improve the system recall while degrading the precision.

Another improvement can be demonstrated by the following example. Consider the sentence:

*The government of Russia loves the government India.*

MINIPAR would extract the following relations from the above sentence: [love subj government] and [love obj government]. Given these relations it’s impossible to decide which government is which without assigning unique index numbers to each of the instances of the word “government”.

### 6.5 Pronoun Resolution

Traditional IR systems treat pronouns as function words and therefore disregard them even though the recall and precision of an IR system would significantly improve if it could replace these pronoun references with its antecedents. Currently SIREN doesn’t have any mechanism that would do that, but SIREN’s framework allows a smooth integration of such a module and adding it would also have a big impact on the performance of the system.
Appendix A

Setting up a New Corpus

The following steps will guide you through the process of indexing a new corpus. For abbreviation "base" directory stands for /projects/infolab/src/Siren.

1. Assuming the whole corpus is contained in one file, the first thing to do would be to put that file into a format suitable for indexing. In order to do that, you should strip the corpus file of all the extra tags that don't need to be indexed (ie. html) and make it such that each article (or any other text unit) is separated by an empty line. For the example's sake let us name this new file newcorpus.clean.

2. Create 2 new subdirectories in /base/corpuses/ and /base/indices/. The names of the subdirectories should be the same as the corpus name (ie. /base/corpuses/worldbook for the worldbook corpus)

3. /base/corpuses/utils/tagCorpus.scr newcorpus.clean newcorpus.tagged
   Tokenizes the new corpus as well as tags each word with its part of speech. Place newcorpus.tagged into /base/corpuses/newcorpus subdirectory.

4. /base/corpuses/utils/stripTags.pl newcorpus.tagged newcorpus.sentence
   Creates a file where each sentence is on a separate line and each article is separated by a new line. The file will be used as an input to the MINIPAR parser.

5. /base/corpuses/utils/parse < newcorpus.sentence > newcorpus.minipar.relations
   Extracts all the relations from the new corpus using the MINIPAR parser as well as
adds in article and sentence numbers for each relation. Place newcorpus.minipar.relations into /base/corpuses/newcorpus subdirectory.

At this point we are ready to create the keyword and relation indices for the new corpus. All of indices will be placed in /base/indices/newcorpus subdirectory.

6. /base/src/solaris/make_index newcorpus
   Creates the keyword based index for the new corpus.

7. /base/src/solaris/make_index -r newcorpus
   Creates the relation index for the newcorpus.

To start SIREN with the newly indexed corpus, run the following:

/base/src/solaris/runir newcorpus
Bibliography


