The INQUERY Retrieval System

James P. Callan, W. Bruce Croft, and Stephen M. Harding

Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003, USA

croft@cs.umass.edu

Abstract

As larger and more heterogeneous text databases become available, information retrieval research will depend on the development of powerful, efficient and flexible retrieval engines. In this paper, we describe a retrieval system (INQUERY) that is based on a probabilistic retrieval model and provides support for sophisticated indexing and complex query formulation. INQUERY has been used successfully with databases containing nearly 400,000 documents.

1 Introduction

The increasing interest in sophisticated information retrieval (IR) techniques has led to a number of large text databases becoming available for research. The size of these databases, both in terms of the number of documents in them, and the length of the documents that are typically full text, has presented significant challenges to IR researchers who are used to experimenting with two or three thousand document abstracts. In order to carry out research with different types of text representations, retrieval models, learning techniques, and interfaces, a new generation of powerful, flexible, and efficient retrieval engines needs to be implemented. At the Information Retrieval Laboratory in the University of Massachusetts, we have been developing such a system for the past two years. The INQUERY system is based on a form of probabilistic retrieval model called the inference net. This model is powerful in the sense that it can represent many approaches to IR and combine them in a single framework [8]. It also provides the ability to specify complex representations of information needs and compare them to document representations.

In this paper, we focus on the architecture and implementation of the INQUERY system, which has been designed for experiments with large databases. We start by giving a brief description of the underlying inference net model in the next section. We then present an overview of INQUERY's architecture, followed by more detailed descriptions of the important system components. Throughout this description, we will give timing figures from recent experiences with a 1 Gigabyte database that contains nearly 400,000 documents varying in length from short abstracts to 150 page reports. We conclude by discussing the current research and development issues.

2 The Inference Network Model

Bayesian inference networks are probabilistic models of evidential reasoning that have become widely used in recent years [1; 6]. A Bayesian inference network, or *Bayes net*, is a directed acyclic graph (DAG) in which nodes represent propositional variables and arcs represent dependencies. A node's value is a function of the values of the nodes it depends upon. Leaf nodes typically represent propositions whose values can be determined by observation. Other nodes typically represent propositions whose values must be determined by inference. The notable feature of Bayes nets is that dependencies are not necessarily absolute. Certainty or probability can be represented by weights on arcs.

INQUERY is based upon a type of Bayes net called a *document retrieval inference network* [9; 8]. A document retrieval inference network, or *inference net*, consists of two component networks: one for documents, and one for queries (see Figure 1). Nodes in an inference net are either *true* or *false*. Values assigned to arcs range from 0 to 1, and are interpreted as belief.

2.1 The Document Network

A document network can represent a set of documents with different representation techniques and at varying levels of abstraction. Figure 1 shows a simple document network with two levels of abstraction: the document text level d, and the content representation level r. Additional levels of abstraction are possible, for example audio or video representations, but are not currently needed by INQUERY.

A document node d_i represents the proposition that a document satisfies a user query. Document nodes are assigned the value *true*. The value on an arc between a document text node d_i and a content representation node r_k is the conditional probability $P(r_k|d_i)$. A document's prior probability $P(d_i)$ is 1/(number of documents).

A content representation node r_k represents the proposition that a concept has been observed. The node may be either *true* or *false*. The value on an arc between a content representation node r_k and a query concept node c_l is the belief in the proposition.

INQUERY uses several types of content representation nodes. The simplest corresponds to a single word of the document text, while more complex concepts include numbers, dates, and company names. Section 4 describes in more detail the types of content representation nodes created, and the methods used to create them.

2.2 The Query Network

The query network represents a need for information. Figure 1 shows the network for a query with two levels of abstraction: the query level q, and the concept level c. Additional levels of abstraction are possible, but are not currently needed by INQUERY.

Query nodes represent the proposition that an information need is met. Query nodes are always *true*. Concept nodes represent the proposition that a concept is observed in a document. Concept nodes may be either *true* or *false*.

The query network is attached to the document network by arcs between concept nodes and content representation nodes. The mapping is not always one-to-one, because concept nodes may define concepts not explicitly represented in the document network. For example, INQUERY's *phrase* operator can be used to define a concept that is not represented explicitly in the document network. The ability to specify query concepts at run-time is one of the characteristics that distinguishes intelligent information retrieval from database retrieval.

2.3 The Link Matrix

Document retrieval inference networks, like the Bayes networks from which they were derived, enable one to specify arbitrarily complex functions to compute the belief in a proposition given the beliefs in its parent nodes. These functions are sometimes called *link matrices*. If the belief for each combination of evidence were specified directly, a link matrix for a node with n parents would be of size $2 \cdot 2^n$. This problem can be avoided by restricting the ways in which evidence is combined. INQUERY uses a small set of operators, described in Section 6, for which closed-form expressions can be found.

3 Overview of the Architecture

The major tasks performed by the INQUERY system are creation of the document network, creation of the query network, and use of the networks to retrieve documents. The document network is created automatically by mapping documents onto content representation nodes, and storing the nodes in an inverted file for efficient retrieval. Query networks are specified by a user through a user interface. Document retrieval is performed by using recursive inference to propagate belief values through the inference net, and then retrieving documents that are ranked high-

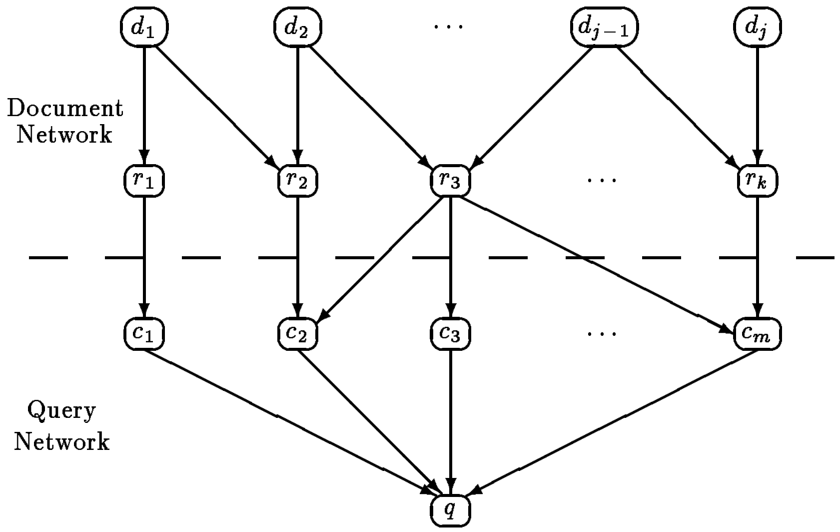

Figure 1: A simple document retrieval inference network.

est. Figure 2 shows the major components of the INQUERY system, and how information flows between them. The following sections discuss each component in more detail.

4 The Parsing Subsystem

The first task in building a document network is to map each document onto a set of content representation nodes. This mapping process is referred to as *parsing* the document, and consists of five components: lexical analysis, syntactic analysis, concept identification, dictionary storage, and transaction generation. It is important that each of these components be efficient, because construction of the document network is one of the most time-consuming parts of building and using inference nets. The current set of INQUERY parsers, without high-level concept recognition, require 19.8 CPU hours on a Sun SPARCserver 490 with 128 MBytes of memory to parse a 1 GByte document collection. The following subsections describe how each of the parsing components is implemented.

4.1 Lexical and Syntactic Analysis

There are three distinct uses of lexical analysis in INQUERY. The *parser's lexical analyzer* provides lexical tokens (usually words or field markers) to the syntactic analyzer. The *database builder* stores the document text in a database for use by the user interface. *Concept analyzers* identify higher-level concepts, for example dates and names, that occur in the text. The activities of these lexical analyzers are loosely coordinated by a lexical analysis manager.

One reason that it is desirable to have so many lexical analyzers is that INQUERY currently contains parsers for six different document formats. The burden of supporting many document formats is minimized by keeping the database builder and concept analyzers ignorant of the document format. The lexical analysis manager enforces this ignorance by controlling access to the input stream. The manager reads large blocks of text into an internal buffer, from which the lexical analyzers read. When a new document is encountered, the parser's analyzer is given exclusive access to the document, as shown in Figure 3a. The parser's analyzer is responsible for converting into canonical format all field markers found in the document. When the parser's analyzer reaches the end of the document, the other analyzers are given access to the document, as shown in Figure 3b.

The parser's analyzer has two important duties besides converting the document to canonical for-

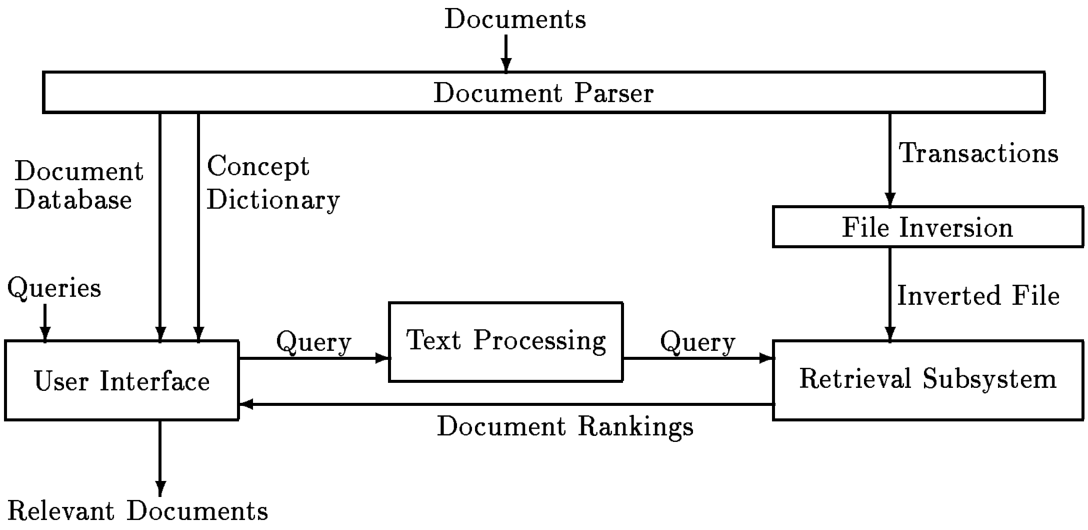

Figure 2: The architecture of the INQUERY information retrieval system.

mat. It is responsible for providing tokens, usually words, numbers, or field markers, to the syntactic analyzer. It is also responsible for converting words to lower case, discarding user-specified *stop* words (e.g. 'a', 'and', 'the'), and optionally removing word endings (e.g. '-ed', '-ing') before notifying the transaction manager (discussed below) about the occurrence of each word.

The principal use of syntactic analysis in INQUERY is to ensure that a document is in the expected format, and to provide error recovery if it is not. All of INQUERY's syntactic analyzers are created by YACC [3].

4.2 Concept Recognizers

INQUERY is currently capable of recognizing and transforming into canonical format four types of concepts: numbers, dates, person names and company names. INQUERY also contains a concept recognizer to recognize and record the locations of sentence and paragraph boundaries. Concept recognizers tend to be complex [5; 7], so it is desirable to implement them as efficiently as possible. All of INQUERY's concept recognizers are currently finite state automata created by LEX [4]. In principal, it is possible to combine the recognizers into a single finite state automaton, however LEX cannot create automata of the required size.

The number and date recognizers use grammars similar to Mauldin's [5]. The major difference is INQUERY's use of string arithmetic to avoid roundoff errors in the number recognizer. The recognizers map different expressions of a concept (e.g. 1 million, or 1000000, or 1,000,000) into a canonical format.

The company name recognizer is similar to, but less sophisticated than, Rau's [7]. It looks for strings of capitalized words that end with one the legal identifiers that often accompany company names (e.g. "Co", "Inc", "Ltd", or "SpA"). If the company name occurs once with a legal identifier, the recognizer can usually recognize all other occurrences of the name in the document. This strategy performs reasonably well on our test collections.

The person name recognizer uses a strategy similar to the company name recognizer, except that it looks for occupation titles and honorific titles. This strategy performs poorly on our test collections. We are contemplating replacing the current algorithm with one that relies more heavily on a large database of known names.

The sentence and paragraph boundary recognizer is currently only able to recognize boundaries that are explicitly tagged with field markers. The locations of these boundaries is saved in a file, for use in a planned project on paragraph- and sentence-level retrieval from large documents

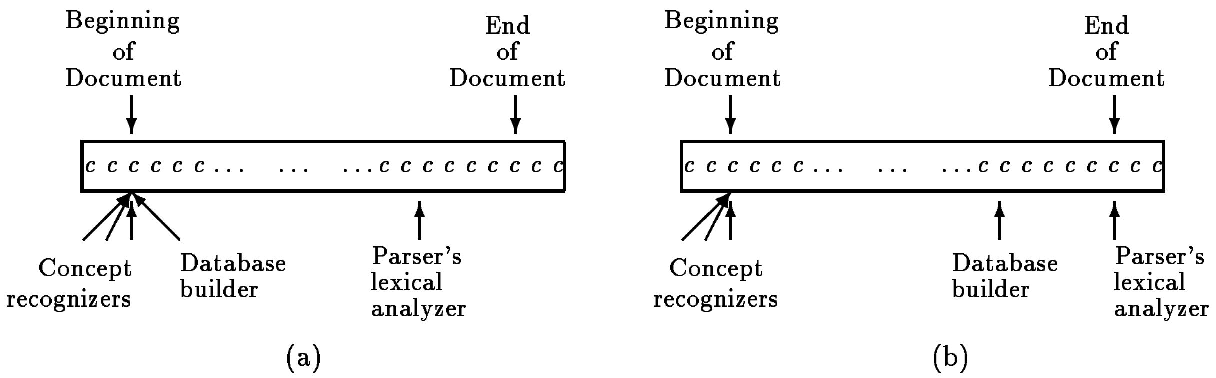

Figure 3: (a) The parser's lexical analyzer converts field markers to canonical form and provides tokens to the syntactic analyzer. (b) When the parser analyzer reaches the end of the document, the database builder and concept recognizers are allowed to read the document, one at a time.

collections.

In principle, there is no limit to the number and complexity of concept recognizers that can be added to INQUERY. For example, we are investigating the use of stochastic tagging [2] to automatically identify phrases. The main consequence of additional concept recognizers is the overhead that they add to the parsing process. The current set of recognizers slows parsing by about 25%.

4.3 Concept Storage

The lexical analyzers are designed to work efficiently with strings of characters, but the rest of INQUERY is not. When a decision is made to index a document by a word or higher-level concept, the string of characters is replaced with its entry number in a *term* dictionary. A reference to an entry number take less space and can be manipulated much more efficiently than a reference to a string of characters. If the word already exists, the number of the existing entry is returned, otherwise a new entry is created.

INQUERY originally stored its dictionary in a B-tree data structure. However, performance analysis showed the dictionary to be a bottleneck. The current version of INQUERY stores its dictionary in a hash table. This change alone reduced the time required to parse a 339 MByte document collection from 16.8 CPU hours to 8.2 CPU hours.

4.4 Transaction Generation

Each time a term is identified, whether by the parser's lexical analyzer or a concept recognizer, its location is reported to the transaction manager. When the end of the document is reached, the transaction manager writes to disk a set of *indexing transactions* that record for each term the frequency and locations of its occurrence in that document.

Transactions are currently stored in text files using a suboptimal encoding method. Our experiments with large collections have produced more transactions than fit on one of our disks. The transaction manager copes with this problem by creating a new transaction file each time an INQUERY document parser is invoked. (One invocation of a parser may parse many documents.) The periodic creation of new transaction files enables us to scatter them across several disks.

5 File Inversion

Each transaction represents a link between a document node and a content representation node. The entire document network is represented by the set of transaction files produced during parsing. The task addressed after parsing is to organize the network so that evidence may be propagated through it rapidly and efficiently.

The value of an internal network node is a function of the values assigned to its parents. INQUERY avoids instantiating the entire document network during retrieval by using recursive inference to determine a node's value. The speed of recursive inference depends upon how fast information about a node and its links can be obtained. INQUERY provides fast access to this information by storing it as an inverted file in a B-Tree data structure.

The inverted file is constructed most efficiently if the transactions for a term are processed together. Therefore the transaction files are sorted before the inverted file is constructed. The sorting procedure involves several steps, both for efficiency and because transactions may be stored in multiple files that do not all fit on one disk.

We begin by using the UNIX sort program to sort each transaction file by term and document identifiers. If the transactions all fit on a single disk, we merge-sort the sorted transaction files. Otherwise we partition the sorted transaction files at some term (e.g. the 10,000th) and merge-sort partitions covering the same ranges of terms.

Sorting is one of the most time-consuming tasks in building a document network. In tests with a 1 GByte document collection, sorting, partitioning and merge-sorting 1.3 GBytes of transactions required 13.6 CPU hours on the Sun SPARC-server 490.

After the transactions are sorted, the inverted file can be constructed in $O(n)$ time. The keys to the inverted file are term ids. The records in the inverted file store the term's collection frequency, the number of documents in which the term occurs, and the transactions in which the term occurs. The inverted file is stored in binary format, which makes it smaller than the transaction files from which it is assembled. The 1.3 GBytes of transactions referred to above were converted to an 880 MByte inverted file in 2.5 CPU hours.

6 The Retrieval Subsystem

The retrieval subsystem converts query text into a query network, and then evaluates the query network in the context of the previously constructed document network.

6.1 Building a Query Network

Queries can be made to INQUERY by using either natural language or a structured query language. Natural language queries are converted to the structured query language by applying the #sum operator to the terms in the query. Table 1 describes #sum and the other operators in INQUERY's query language. Query operators permit the user to provide structural information in the query, including phrase and proximity requirements. Query text is converted to lower case, possibly checked for stopwords or stemmed to canonical word form, and compared to the concept dictionary before being converted into a query net.

Query net nodes correspond to structured language operators and query terms. The information contained in a node varies, depending on its type. The attachment of the query net to the pre-existing document net occurs at the term nodes.

6.2 Retrieval Engine

The INQUERY retrieval engine accepts the root node of a query net and evaluates it, returning a single node containing a belief list. This belief list is a structure containing the documents and their corresponding "beliefs" or probabilities of meeting the information need as defined by the query. The retrieval engine does its work by instantiating proximity lists at term nodes, and converting such lists to belief lists as required by the structure of the query net, using methods defined in [9]. This list may be sorted to produce a ranked list of documents for the user to see.

The inference net is evaluated by recursive calls of the main evaluation routine which in turn calls one of many possible node specific evaluation routines. The routines represent the canonical form of evaluating a simplified link matrix at each node. The closed form expressions for computing the belief at node Q are:

$$\begin{aligned}
\mathrm{bel}_{\mathrm{not}}(Q) &= 1 - p_1 && (1)\\
\mathrm{bel}_{\mathrm{or}}(Q) &= 1 - (1-p_1)\cdot \ldots \cdot (1-p_n) && (2)\\
\mathrm{bel}_{\mathrm{and}}(Q) &= p_1 \cdot p_2 \cdot \ldots \cdot p_n && (3)\\
\mathrm{bel}_{\mathrm{max}}(Q) &= \max(p_1, p_2, \ldots, p_n) && (4)\\
\mathrm{bel}_{\mathrm{wsum}}(Q) &= \frac{(w_1 p_1 + w_2 p_2 + \ldots + w_n p_n) w_q}{(w_1 + w_2 + \ldots + w_n)} && (5)\\
\mathrm{bel}_{\mathrm{sum}}(Q) &= \frac{(p_1 + p_2 + \ldots + p_n)}{n} && (6)
\end{aligned}$$

The basic structures from which all computations of document node belief are derived are proximity lists and belief lists. A proximity list contains statistical and proximity (term position) information by document on a term specific basis. The belief list is a list of documents and associated

OPERATOR	ACTION
#and	AND the terms in the scope of the operator.
#or	OR the terms in the scope of the operator.
#not	NEGATE the term in the scope of the operator.
#sum	Value is the mean of the beliefs in the arguments.
#wsum	Value is the sum of weighted beliefs in the arguments, scaled by the sum of the weights. An additional scale factor may be supplied by the user.
#max	The belief is the maximum of the beliefs in the arguments.
#*n*	A match occurs whenever all of the arguments are found, in order, with no more than *n* words separating adjacent arguments. For example, #3 (A B) matches "A B", "A c B" and "A c c B".
#phrase	Value is a function of the beliefs returned by the #3 and #sum operators. The intent is to rely upon full phrase occurrences when they are present, and to rely upon individual words when full phrases are rare or absent.
#syn	The argument terms are to be considered synonymous.

Table 1: The operators in INQUERY's query language.

belief values at a given node, as well as default beliefs and weights used when combining belief lists from different nodes. The belief list will contain the cumulative probability of a documents' relevance to the query given the values of the parents. Belief lists may be computed from proximity lists, but the reverse derivation is not possible. This limitation imposes some restrictions on query form. The query form must not produce a proximity list type resultant node which is acted upon by a routine expecting a belief list type. Proximity lists are transformed into belief values using the information in the list and combined using weighting or scoring functions.

Node belief scores are calculated as a combination of term frequency (tf) and inverse document frequency (idf) weights. The values are normalized to remain between 0 and 1, and are further modified by tf and belief default values which the user may define at program invocation.

Calculation of a belief for a given node is dependent on the type of node and the number and belief in its parents as presented in Equations 1-6. The probability combinations are achieved via belief list merges and negation.

6.3 Retrieval Performance

Typical query processing time is 3 to 60 seconds on a 1 GByte document collection. Processing time varies according to query complexity, the number of terms in the query and their frequency in the collection. Terms with high collection frequencies are likely to add to processing time due to the length of associated proximity lists. Retrieval performance is much improved over boolean and conventional probabilistic retrieval. The reader is referred to [9] for details.

7 Interfaces

INQUERY offers batch and interactive methods of query processing, and an application programmers interface (API) to support development of customized front-ends to the retrieval engine. Each of these interfaces is discussed below.

7.1 Application Programmers Interface

The INQUERY application programmers interface (API) is a set of routines that allow programmers to develop interfaces of their own to the INQUERY retrieval engine. The API functions open and close INQUERY databases and files, convert query text into query nets, evaluate query nets, and retrieve documents.

7.2 Batch Interface

The batch program takes command line arguments in the form of input file names and switches. The output of the program is a ranked list of documents by weight (the calculated probability of relevance) in a file format readable by an evaluation program, which can produce standard recall-precision tables on retrieval performance. A file of relevance judgments for the submitted queries is required as input for the batch program. This arrangement allows queries to be run repeatedly, so that changes to the system may be evaluated.

7.3 User Interface

The interactive user interface supports queries in natural language or structured form, and was produced using routines from the API. Query results are displayed on the screen in the form of a ranked document list. The user may browse through the retrieved documents to determine their relevance to the query. A file containing the session results may also be produced.

8 Current Status

The INQUERY system has been tested on both standard information retrieval collections [9; 8] and a heterogeneous 1 GByte collection. We continue to conduct research on intelligent information retrieval with the INQUERY system, and encourage others to do so. INQUERY version 1.3, described in this paper, is distributed by a technology transfer agency of the University of Massachusetts for a nominal fee.

Current work on INQUERY addresses both software engineering and research issues. One recent improvement was the addition of encoding methods to reduce the sizes of both the inverted file and the user-interface indices. The inverted file index has been reduced to 40% of its previous size, while the user-interface index has been reduced to 5% of its previous size. This improvement will enable us to install a 2 GByte document collection on our current hardware during the summer of 1992.

We are also studying the use of relevance feedback in INQUERY. Relevance feedback enables a user to identify those retrieved documents that are most relevant to the user's information need. The system then analyzes those documents, produces a revised query based upon the analysis, and retrieves a new set of documents.

A colleague is developing a Japanese version of INQUERY, called JINQUERY. The only differences between INQUERY and JINQUERY are the lexical and syntactic analyzers, and the user interface. Japanese documents are particularly challenging because word boundaries are implicit. JINQUERY currently indexes documents with Kanjii characters and Katakana words. A segmenter that divides a stream of Kanjii characters into words is being tested.

Finally, research is underway to provide better support for queries expressed in natural language. INQUERY and JINQUERY currently handle natural language by summing the beliefs contributed by the individual query words. We believe that improvements can be made by automatically identifying phrases, incorporating words from thesauruses, running the concept recognizers on queries, and performing other types of morphological processing.

Acknowledgements

This research was supported in part by the Air Force Office of Scientific Research under contract AFOSR-91-0324.

References

[1] Eugene Charniak. Baysian networks without tears. *AI Magazine*, 12(4), Winter 1991.

[2] Kenneth Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the 2nd Conference on Applied Natural Language Processing*, pages 136–143, 1988.

[3] Stephen C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual.* Bell Telephone Laboratories, Inc, Murray Hill, NJ, 1979.

[4] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. In *UNIX Programmer's Manual.* Bell Telephone Laboratories, Inc, Murray Hill, NJ, 1979.

[5] Michael Loren Mauldin. *Information retrieval by text skimming.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, 1989.

[6] Judea Pearl. *Probabilistic reasoning in intelligent systems: Networks of plausible inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[7] Lisa F. Rau. Extracting company names from text. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, 1991.

[8] Howard Turtle and W. Bruce Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3), July 1991.

[9] Howard R. Turtle and W. Bruce Croft. Efficient probabilistic inference for text retrieval. In *RIAO '91 Conference Proceedings*, pages 644–661, Barcelona, Spain, April 1991.