# Relevancy Ranking of Web Pages Using Shallow Parsing

**Yuval Z. Feinstein**
**Claudia V. Goldman** *
**Yishay Mor**
**Jeffrey S. Rosenschein**
Institute of Computer Science
The Hebrew University
Givat Ram, Jerusalem, Israel
{yuvalf,clag,yish,jeff}@cs.huji.ac.il

**Abstract**

We present a Phrase-Structure-Grammar method to rank the relevancy of Web pages.

This method might be used as an aid to an information agent, assisting it in understanding the information acquired. The method could guide the agent when evaluating text by taking into account the syntactic roles of the words in this text. In cases where the agent presents its user with results from a search, our system could also assist this agent by ranking the results according to their relevance to the user's query.

An existing search engine can also take advantage of this kind of method to improve the precision of its search results, without undermining recall.

In both cases, we focus on ranking the pages retrieved for the user based on the relevance of these pages to the user's query. Therefore, we expect that our method will decrease the need to browse through irrelevant pages.

The system is based on a syntactic analysis that requires the construction of parse trees for part of the text. Nevertheless, we combine shallow parsing with heuristics, thus making the system's computational requirements practical for real-world applications.

## 1 Introduction

Typical web search engines today receive a query composed of several keywords, and return a list of Web pages that contain any of the keywords (an "OR" search) or all of them (an

*Supported by the Eshkol Fellowship, Israeli Ministry of Science

"AND" search), disregarding their relative order or importance. As a result, a user may spend many minutes going through irrelevant retrieved pages before reaching a relevant document.

The standard measures of performance of an information retrieval system [SM83] are *recall* and *precision*. Let $N$ be the total number of pages relevant to the query, $M$ be the number of pages retrieved, and $P$ be the number of relevant pages retrieved.

**recall** is the ratio of relevant pages retrieved out of all the relevant pages, that is $P/N$.

**precision** is the ratio of relevant pages retrieved out of the total number of retrieved pages, that is $P/M$.

We suggest increasing precision by using a more elaborate analysis of the user's query. Given a query (preferrably a simple noun phrase like "a red herring"), we try to find relevant WWW pages, and rank them according to their relation to the query. By *relevance* we mean how close the retrieved page is to the meaning of the query.

Our emphasis is on precision rather than recall (i.e., we may not get all relevant pages, but intend that every page we get is relevant — the most relevant first, the second afterwards, etc.). In the current implementation, our aim is to rank the pages retrieved by an existing Web search engine in decreasing order of relevancy.

When we use a Web search engine (or any information retrieval tool), we may get several documents that meet our query criteria. *Relevancy ranking* is a process that tries to order the documents according to their relevance to the user's needs. There are several heuristics used for relevancy ranking in existing Web search engines. They include: [spi]

1. Search term frequency — if a word from the query appears in the page many times, rank the page high.

2. Some search engines consider both the *frequency* and the *positioning* of keywords to determine relevancy, reasoning that if the keywords appear early in the document, or in the headers, this increases the likelihood that the document is on target.

3. Number of links to the page ranked — if there are many links from other pages, probably some other people think the page is important. Obviously, to use this heuristic we need to know in advance all the pages we are checking, and therefore all the links from the pages to the set we are ranking.

4. Relevance weights on terms — AltaVista [alt] lets the user assign weights to the query terms, and if a word with a high weight appears more times, the rank is higher.

5. Inverse Document Frequency — Rare terms (within the entire database) receive a higher weight of relevancy — i.e., if a rare word appears both in the query and in the page, rank it higher.

6. Density — The comparable length of retrieved documents has a subtle affect on the weight of relevancy (a longer document has more content, but may have disproportionately more appearances of the search terms).

The common feature of these heuristics is that they use individual words, without assuming any structure on these words. However, in most of the search engines that use any of these heuristics, phrases like "College Junior" and "Junior College" are inseparable, Searching for either of the phrases will retrieve all pages containing both words, i.e., all pages containing any of the two phrases. Worse still, looking for "Blue Moon" using these heuristics may retrieve all pages containing both "Blue" and "Moon", irrespective of their relative places or order. Using a typical search engine we may still get a list of irrelevant pages before we get a relevant one.

In [EZ96], Evans & Zhai propose using noun-phrase extraction to increase recall and precision in information retrieval. Their claim is that using noun-phrases as index terms will improve current database retrieval performance. They have used statistical noun-phrase extraction to identify *concepts* that are likely to be recalled. Using a list of the concepts they improved both recall and precision of responses to queries.

In the research described in the current paper, we have used shallow parsing, including noun phrase identification, to rank relevancy of searched Web pages according to a given query. *Shallow Parsing* means finding a parse tree generating our phrase, without going into the semantic meaning of the phrase or using precise agreement constraints (e.g., we don't check whether all parts of our phrase agree on number).

We parse the text online, rather than prepare a prior noun phrase index, as done in [EZ96]. The only information we save is the syntactic head and a parse tree representing the query. In addition, our system ranks the relevant documents such that the user will see the most relevant documents first, and all the other documents in decreasing order of relevance.

This paper is organized as follows. The system will be described in Section 2. Later, The results obtained appear in Section 3, and conclusions and possible uses of the technique appear in Section 4.

# 2 The System

Our research uses noun-phrase parsing and comparison for relevancy ranking of retrieved Web pages. Our system is based on a combination of three techniques: syntactic parsing, heuristics, and statistics. Our goal is to let the user see the more relevant pages first.

This method might be used as an aid to an information agent, assisting it in understanding the information acquired. The method could guide the agent when evaluating text by taking into account the syntactic roles of the words in this text. In cases where the agent presents its user with results from a search, our system could also assist this agent by ranking the results according to their relevance to the user's query.

An existing search engine can also take advantage of this method to improve the precision of its search results, without undermining recall.

Our agent receives a query, composed of a list of words. It calls a search engine with the words in the query. The query is syntactically parsed. The agent compares each page retrieved with the parsed query, by parsing parts of the retrieved pages (i.e., text chunks). The agent calculates the distance between the trees of the parsed parts and the query tree, and gives each page a relevancy grade. The output is a list of the links in the order of their relevancy.

## 2.1 The Algorithm

Our system receives as input a phrase (preferably a noun phrase), composed of several words. Most search engines treat the query simply as a list of words that have to appear somewhere in the page. We look for a syntactic element — the query has to be a phrase in order to initiate a search. If the query is a phrase it is an advantage because we can use shallow parsing to identify the important words, and rank pages accordingly. *Shallow Parsing* (as mentioned above) means finding a parse tree generating our phrase, without going into the semantic meaning of the phrase or using precise agreement constraints (e.g., we don't check whether all parts of our phrase agree on number).

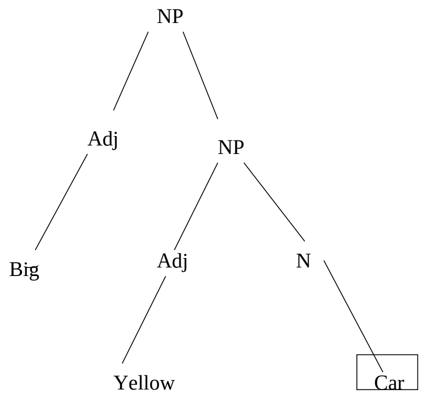


Figure 1: A syntactic head

### 2.1.1 Query Handling

1. The query is parsed using a Prolog-based parser (see Section 2.2 for further details).

2. The main objective of the parsing is to find *the syntactic head*. Every phrase (and sentence) has a single head, which can be determined from the structure of the phrase. For example: the head of the NP "Big Yellow Car" (see Figure 1) is the noun "Car" — which is the basic word in the phrase, while "Yellow" and "Big" are adjuncts. The head of the phrase "John went to see Mary" is the verb "went". We use two properties of the head:

- A head is more important than the nearby words for the meaning sought (this is an issue under debate among linguists, but we assume it for now).
- A head can be determined from the rewrite rules during the parse.

3. If the parsing was successful, the syntactic head and tree resulting from the parsing process are saved for later use. The syntactic head is found by saving local head information in the tree. If the parsing is unsuccessful, the agent will not improve on the search engine's result.

4. We give the content words in the query to a search engine (we used the Altavista/Yahoo search [yah]), to get a list of Web pages that contain relevant words. Stop words are not counted as relevant. The search engine uses an "OR" based query — if a single content word is in the page, it should be returned. We cannot rank pages which the search engine did not return in the first place, as our program does not have another search module.

#### 2.1.2 Page Evaluation

Each WWW page returned from the search engine is evaluated using the following steps:

1. We first analyze the text using a finite automaton. The automaton is used to accumulate text according to HTML tags. Text chunks are formed according to the following heuristics:

   - "important" HTML tags, title and headers, cause a chunk to be created. for example: <h1>some header</h1> will send the chunk "some header" for evaluation, disregarding what comes before and after it. Several important HTML tags are: <h1>,<h2>,... (headers)
     <title>.
   - the text after "unimportant" HTML tags, i.e., all the other HTML tags, is appended to the current chunk, until an end of a paragraph is met. Then, the whole paragraph is evaluated.

2. Since we have not parsed the chunks yet, every chunk that includes *any* part of the query, even in a different order, and even if the words do not follow one another, is sent to the parser.

3. Each such text chunk is evaluated by trying to parse it, in a manner similar to the query parsing (see 2.1.1). If the parse succeeds, the head and tree of that chunk are compared to the corresponding head and tree of the query, as explained in Subsection 2.2.

4. For each page, a total result is returned by weighting HTML tags (see item 1) and text (see item 3). The automaton assigns to each text chunk a *weight* according to the HTML tags met while accumulating it (for example, a text chunk near a "title"

HTML tag gets weight 0.5, while a text chunk with no important HTML tags nearby gets weight 0.1.)

All pages are sorted according to their relevance grade, and the pages' titles and URLs are printed in the corresponding order.

## 2.2 Implementation Details

1. The agent is a patchwork object that uses the following programs:
   - Amir Langer's IA library [lan] that receives a URL as its input, and outputs a data structure listing the links and text included in the corresponding Web page.
   - Eric Brill's syntactic tagger [Bri94] receives a chunk as input, and gives syntactic tags (Part Of Speech) to each word. For example, the chunk "The big bad Wolf" would be tagged as: The/DT big/JJ bad/JJ Wolf/NNP meaning determiner, adjective, adjective and noun respectively.
   - Our Prolog parser that receives the tags as input, and tries to create a tree with the tags as its leaves.
2. For the parsing we do on the query and on the document text, we use a simple Definite Clause Grammar (DCG), which becomes an LL parser using Prolog's search order. The leaves in the trees are tags obtained from Brill's tagger.

   The parsing process finds the syntactic head [Hae91] using a local pointer — for each grammar rule, we indicate whether the head will be in the right subtree or in the left subtree of the tree produced from an application of that rule, so that a path from the tree's root to the head is saved in the tree.

   The grammar is shown here:

   - General rules:
     np $\longrightarrow$ det np1
     np $\longrightarrow$ n
     np $\longrightarrow$ adj np
     pp $\longrightarrow$ prep np
     np $\longrightarrow$ np1 pp
     np $\longrightarrow$ n nlist
     nlist $\longrightarrow$ n
     nlist $\longrightarrow$ n nlist
     vp $\longrightarrow$ vp1 np
     vp $\longrightarrow$ v
     np1 $\longrightarrow$ n
     np1 $\longrightarrow$ det np
     vp1 $\longrightarrow$ v

- Rules involving leaves:
  adv $\longrightarrow$ RB
  dt $\longrightarrow$ DT
  adj $\longrightarrow$ JJ
  n $\longrightarrow$ NN
  n $\longrightarrow$ PRP
  prep $\longrightarrow$ IN
  v $\longrightarrow$ VB

3. Trees are compared in two ways:

   (a) First, their syntactic heads are compared. Suppose the query was "The Big Cliff" and the phrase checked is "A Tall Cliff" — in this case the syntactic heads (i.e., Cliff) are the same. If the syntactic heads are the same, the trees are similar, and this can be extended to include word classes (i.e., synonyms), etc. If the phrase on the page is "Cliff Hanger," the syntactic heads are different, which means the phrase is further apart from the query. Even if the heads are different, the trees are compared as explained in item 3b.

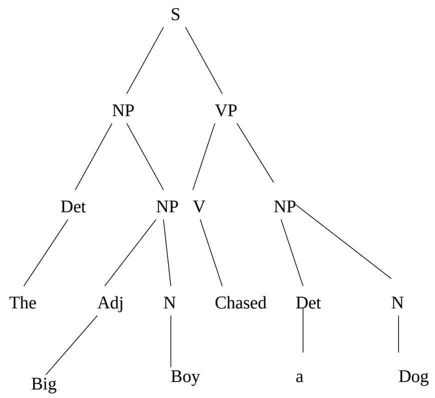


Figure 2: Tree Representation of a sentence

   (b) Now we compare the tree that represents the text chunk ($T_w$) with the tree representing the query ($T_q$). Each tree is represented as a linearized branch list [Ofl96]. Each branch is represented as a list of the node labels, from the leaf upwards. The linearized representation of the tree in Figure 2 is:

   - The Det NP S
   - big Adj NP NP S
   - boy N NP NP S
   - chased V VP S
   - a Det NP VP S

- Dog N NP VP S

(c) Individual branches are compared as follows: if a branch in $T_w$ has a leaf identical to one of the leaves in $T_q$, their linearized branches are checked for minimal edit-distance (Levenshtein measure) [Man89], and an inverse function of the distance is added to the page's rank. *Levenshtein measure* is the minimal number of letter insertions, deletions and substitutions needed to get from string A to string B. The "strings" here are the branch representations, and the "letters" are the node labels.

An example: suppose the ($T_q$) branch is (from the leaf up): blue-JJ-adj-np-np
and the ($T_w$) branch is: blue-JJ-adj-np-np-np
then the edit distance (that we will call $d(q,w)$) is 1 (delete the last "np" from the page branch).

(d) Let the head of the query tree be called $h_q$, and let the head of the chunk tree be called $h_w$. We'll define $\delta_{qw} = 1$ if $h_q = h_w, 0$ otherwise.
Let $\theta_{qw} = \frac{1}{1+d(q,w)}$. To evaluate a chunk, we will combine the two values using the constants $p$ and $q$, while $0 \leq p \leq 1, 0 \leq q \leq 1$, and set $v_i = p\delta_{qw} + q\theta_{qw}$ ($v_i$ is the value of chunk i). The relative values of $p$ and $q$ show how important we think the influences of the head and the rest of the tree are, respectively (for the experiment here, we used 0.5 as both $p$ and $q$).

(e) The final ranking of the page is a weighted sum of all the chunk comparisons, where the weights come from the tags corresponding to text chunks ("title" weighs 0.5, "p" weighs 0.1, etc.). Letting $n$ be the number of chunks parsed, $v_i$ be the similarity value between chunk $i$ and the query tree, and $w_i$ be the weight of chunk $i$, then the page value is:

$$P = \sum_{i=1}^{n} w_i v_i$$

## 2.3 Complexity

Given a query, our system starts by tagging the words in it and then continues by parsing it. Then, for any Web page the system is asked to evaluate based on the given query, the following steps are performed. First, we divide the text in the document into chunks. Chunks that do not include any word from the query are discarded. The words in the remaining chunks are tagged. These chunks are parsed, and their trees are compared to the tree built out of the query.

The complexity of the process of tagging a word in a sentence is the time it takes to find the word in a dictionary, i.e., the time is $O$(length of the word). Parsing a string of length $n$ takes $O(n^3 \times SizeOfGrammar)$ [Ear70]. In our case the grammar is small, and $n$ is bounded. In one case, $n$ denotes the length of the query which is in practice generally quite small. We might even bound it in our implementation to a fixed maximal length. In the second case, $n$ denotes the length of the chunk of text including any word or words from the query. Also, here we might bound it in the implementation. In practice, these chunks are not overly long (e.g., 40 words might be the maximal number of words in a relevant chunk).

At the end, we perform a comparison between the tree of a chunk of text, and the tree of the query. This requires finding equal leaves in both trees and then computing the Levenshtein distance between the corresponding branches. This can be done in $O(m \times n)$, where $m$ is the length of the first branch and $n$ is the length of the second. In our case, the depths of the parse trees are bounded by the number of words in the query ($|Q|$) and in the chunk ($|C|$). Thus, for each word in the chunk we have a computation of complexity $O(|Q| \times |C|)$, and for each chunk $O(|Q|^2 \times |C|)$.

In summary, the complexity of the system presented is dominated by the complexity of the parsing process that is $O(n^3)$, where $n$ is to be taken as the maximum among the length of the query and the maximal length of a relevant chunk of text. The query's length is defined by the user, and is not likely to be long. As we mentioned above, we can produce valuable results even when the chunk size is limited to a fixed bound. Given these two facts, the algorithm's complexity is quite reasonable. For all practical purposes the complexity is linear in the input length, and the algorithm can be implemented as an on-line filter for boosting search results.

# 3 Results

We tested the system using several simple query phrases.

On the query "Glass Eye", we got a ranking which seperated "Glass Eye" from "Eye Glasses" (similar words, different syntactic heads), a separation that is reasonable.

On the query "Distributed Artificial Intelligence", we got a list of relevant pages, even though our parser misparsed the phrase and took "Distributed" to be the head (where the head should be Intelligence).

On the query "A Long Sad Story", the ranking was also plausible, with pages named: "A Sad Story", and "A Sad Short Story" leading the rank.

However, when the desired phrase is in the middle of a chunk of text, or the query is less specific, the performance deteriorates. In response to "Software Agents" our program, taking "Agents" as the syntactic head, ranked many insurance and real-estate agents higher than software agents.

The program typically returns meaningful ranking of about half of the documents in the Yahoo list (most of the others cannot be retrieved — either the URLs are outdated or the servers were down). No unrelated links were ranked highly (i.e., no "false positives"). However, a few pages which contain a desired concept were not ranked high enough. Also, in these cases, the program was more sensitive when a phrase similar to the query appeared at the beginning of a chunk of text, and less sensitive when the query appeared in the middle of the chunk.

Some possible sources for misranking:

1. The grammar used was small, generating few possible phrases.

2. Not using a window — if a phrase appears in the middle of a chunk of text, it may be overlooked.

3. Not using synonyms, or word classes.

# 4 Conclusions and Further Work

We started the project in order to check whether better syntactic analysis of pages would reduce the need to browse through irrelevant Web pages. We hoped to improve ranking by several percentage points. The benefit of using syntax to help ranking and retrieval is to get closer to a concept than by using a word list. Adding our program to a search engine should generally improve its results, as we don't throw away any page retrieved, but only try to rank them better.

Our agent succeeded in performing the phrase-comparing process between query and pages. Our results show that the ranking is not sufficient for independent use, i.e., for using our program as a search engine. However, the syntactic information extracted by this system might improve the heuristics that a learning agent, such as "MUSAG" [GLR96] uses to understand the context of words on the Internet.

Several possible ways to improve performance include using word classes (i.e., semantically close words; whenever looking for "lion" as a head, also look for "tiger" and "puma"), and lemmatizing (i.e., taking a word to its base form; whenever looking for "leaves" look also for "leaf"). These two methods may increase similarity between a query and a page, and thus give a more accurate ranking procedure.

# Acknowledgments

This work has been supported in part by the Israeli Ministry of Science and Technology (Grant 032-8284) and the Israel Science Foundation (Grant 032-7517).

# References

[alt] *altavista*. http://altavista.digital.com/.

[Bri94] E. Brill. Some advances in rule-based part of speech tagging'. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 722–727, Seattle, Wa., 1994.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 14:453—460, 1970.

[EZ96] David A. Evans and Chengxiang Zhai. Noun-phrase analysis in unrestricted text for information retrieval. *The Computation and language E-Print Archive*, 1996. http://xxx.lanl.gov/ps/cmp-lg/9605019.

[GLR96] Claudia V. Goldman, Amir Langer, and Jeffrey S. Rosenschein. Musag: an agent that learns what you mean. In *Proceedings of the First International Conference on Practical Applications of Intelligent Agents and Multi-Agents Technology (PAAM96)*, pages 311–329, London, UK, 1996.

[Hae91] Lilian M. V. Haegeman. *Introduction to Government and Binding Theory.* Basil Blackwell, Oxford, 1991.

[lan] Ialib. Software developed at the Hebrew University by Amir Langer.

[Man89] Udi Manber. *Introduction to Algorithms - A Creative Approach.* Addison Wesley, Reading,Massachusetes, 1989.

[Ofl96] Kemal Oflazer. Error-tolerant tree matching. *The Computation and language E-Print Archive*, 1996. http://xxx.lanl.gov/ps/cmp-lg/9604003.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* Mcgraw-Hill Book Co., New York, 1983.

[spi] The spider's apprentice - how to use web search engines. http://www.monash.com/spidap.html.

[yah] *yahoo.* http://av.yahoo.com/.