Juru at TREC 10 – Experiments with Index Pruning

David Carmel, Einat Amitay, Miki Herscovici, Yoelle Maarek,
Yael Petruschka, Aya Soffer
IBM Labs, Haifa University, Haifa 31905, Israel

1 Introduction

This is the first year that Juru, a Java IR system developed over the past few years at the IBM Research Lab in Haifa, participated in TREC's Web track. Our experiments focused on the ad-hoc tasks. The main goal of our experiments was to validate a novel pruning method, first presented at [1], that significantly reduces the size of the index with very little influence on the system's precision. By reducing the index size, it becomes feasible to index large text collections such as the Web track's data on low-end machines. Furthermore, using our method, Web search engines can significantly decrease the burden of storing or backing up extremely large indices by discarding entries that have almost no influence on search results.

In [1] we showed experimentally, using the LA-TIMES collection of TREC, that our pruning algorithm attains a very high degree of pruning with hardly any effect on precision. We showed that we are able to reduce the size of the index by 35% pruning with a slight decrease in average precision (7%), and with almost no effect on the precision of the top 10 results. For 50% pruning, we were still able to maintain the same precision at the top 10 results. Thereby, we obtained a greatly compressed index that gives answers that are essentially as good as those derived from the full index.

One important issue that was not addressed in our previous work dealt with the scalability of our pruning methods. In this work, we repeat our previous experiments on the larger domain of the Web Track data. While our previous experiments were conducted on a collection of 132,000 documents (476 MB), for the current experiments we built a core index for the entire Wt10g collection, (1.69M documents, 10GB). We then ran our pruning algorithms on the core index varying the amount of pruning to obtain a sequence of pruned indices. Section 4 describes the reuslts of the runs obtained from this sequence of indices. In order to be able to index such a large collection and retreive high quality results, we had to scale Juru's basic indexing and retrieval techniques as well as modify them to specifically handle Web data.

The rest of the paper is organized as follows: Section 2 describes Juru's main functionality. Section 3 briefly describes the pruning algorithms used for the experiments conducted in this work. Section 4 describes the experiments and the results we obtained. Section 5 concludes.

2. Juru's Main Functionality

Juru is a full-text search engine purely written in Java. Juru is based on the Guru search engine described in [2]. The major tasks performed by Juru are: (1) indexing large collections of documents, (2) formulating a query from a free-text query entered by the user, and (3) retrieving relevant documents for a given query and ranking them by order of relevance. In the following sections, we briefly describe the main techniques and algorithms embodied in Juru.

2.1 Profiling

Following the classical inverted index approach, Juru creates an index by associating terms with the documents that contain them, and then storing this mapping in inverted files for efficient retrieval. The first stage for creating this mapping is document parsing which constructs a canonical form called a *document* *profile*. Parsing includes HTML parsing, tokenization, lower case conversion, sentence splitting, and stemming. For HTML parsing Juru uses a parser that extracts from an HTML page the title, the set of out links, and the regular text that appears in the page. For tokenization and sentence splitting Juru uses procedures described in [3]. For stop-word filtering, the system uses a very short default list of stop-words but allows users to define their own special purpose list. Stemming is performed using Porter's stemmer [4]. A default list of proper names is managed by Juru and can be expanded by a special purpose proper name list provided by the user. A term identified as a proper name is not stemmed during profiling. The document profile can be described as a vector of terms, each one associated with its number of occurrences in the document.

For the experiments conducted on the Web Track data we also developed a special description database that provides for each page *p* in the collection a set of descriptions extracted from other pages that cite (i.e., link to) *p*. A description is defined as the anchor text associated with the link. Juru indexes every page based on its content as well as its set of descriptions. Simulations that we conducted on previous Web Track data show that using descriptions as indexing units for HTML pages improves precision by about 20%.

2.2 The indexing process

Documents are indexed as follows: when adding a document to the index, it is assigned an internal unique identifier (*id*). The document *id*, its name, its title, and any additional metadata are stored in a special database called the *document database*. Each term in the document profile is then inserted into a dictionary. The dictionary is managed using a trie data structure. Allow us to remind here that a trie is a search tree, where each node represents a word in a given lexicon or dictionary and each edge is associated with a sequence of characters. Sequencing the characters on the path from the trie root to a trie node composes the word associated with that node. As new words are added to the trie, new nodes are created, and old nodes (containing words) are possibly split.

Each term in the dictionary has a corresponding posting list – a list of all documents where it appears. Each entry in the posting list of term *t* consists of the document *id* of the document containing *t*, the number of occurrences of *t* in *d*, and a list of occurrence offsets of *t* in *d*. The posting lists are stored in a repository called the *repository database* and each term in the dictionary is associated with a pointer to its corresponding posting list. The posting lists are compressed using compression techniques described in [5]. Indexing of the entire collection of documents is carried out in a two-stage process:

Stage 1: creating a forward index:
Each document profile is split into sub-profiles, where each sub-profile contains all terms with a common prefix. For each prefix, all the corresponding sub-profiles from all documents are written to an appropriate forward index file. A forward file for a specific prefix holds for each document a list of all the document terms that begin with that prefix.

Stage 2: inverting the forward index:
After all the document profiles have been split into the forward index files, each file is traversed, using the following algorithm:

```
For each document d in the forward file
    For each term t in d
        If t is found in the dictionary
            retrieve the posting list of t, p(t), from the repository database
        else
            add t to the dictionary
            create a new posting list  p(t)
        add d with all occurrence information to  p(t)
        update p(t) in the repository database
```

The forward index structure allows us to keep only a small part of the dictionary in main memory during indexing, thus enabling index creation with limited memory resources. The dictionary is not required at all during the first stage of indexing, while the second stage requires only the part of the dictionary that corresponds to the prefix of the particular forward file being processed. In addition to restricting the amount of memory needed for indexing, forwarding can be done in parallel on several machines. The forward index technique also speeds up the indexing process by employing effective caching and buffering techniques.

2.3 Query Evaluation

Queries are treated as follows: a query profile is created using the same profiling method that is used for the full document collection during indexing. Recall that a profile is a vector of terms, where each term is associated with the number of occurrences of this term in the document/query. The following algorithm, based on the ranking process of the SMART system [6], describes the ranking process applied by Juru:

```
Input:   a query profile - $q = (t1,t2,..tk)$,
         the number of documents to retrieve - $N$
Output: the $N$ most relevant documents for the input query in document collection $D$

1. For each query term, retrieve its posting list from the repository database
2. Sort the query terms according to the length of their posting lists (handle infrequent terms first)
3. For each $id$ in collection $D$, $Score(id) = 0$
4. For each term $t$ in $q$ with posting-list $p(t)$
         for each posting entry $(d,OccNo(t,d))$ in $p(t)$
                  $Score(id) = Score(id) + tf(t,q) * tf(t,d) * idf(t)$    (*)
5. Normalize document scores by document length $|d|$
   For each $id$ in collection $D$,
                  $Score(id) = Score(id) / |d|$     (**)
6. Return $N$ documents with the highest scores.

(*) term frequency for profile $x$ ($x$ is $d$ or $q$):
                  $tf(t,x) = log(1+OccNo(t,x))/log(1+avg.OccNo(x))$
                        $OccNo(t,x)$ – number of occurrences of $t$ in $x$
                        $AvgOccNo(x)$ – average number of term occurrences in $x$
    inverse document frequency:
                  $idf(t) = log(|D|/|Dt|)$
                        $|D|$ - number of documents in the collection $D$
                        $|Dt|$ - number of documents containing $t$
(**)  document length:
                  $|d| = (0.8*avgDocLength + 0.2*(\# \text{ of unique terms in } d))^{0.5}$
                  $avgDocLength$ – average number of unique terms per document in $D$
```

In order to optimize its query processing time, Juru applies dynamic pruning methods as described in [7]. The main idea is to order the query terms by decreasing weight, according to the length of their posting lists, and to process the infrequent terms first until some stopping condition is met. The algorithm marks each query term as infrequent, frequent, or very frequent. After assigning scores to a sufficient number of documents, very frequent terms are completely ignored. Only posting lists of infrequent terms are fully processed. Frequent terms contribute only to the scores of the documents that have already been scored previously.

2.4 Improving search precision by incorporating lexical affinities

Lexical affinities (*LAs*) were first introduced by Saussure in 1947 to represent the correlation between words co-occurring in a given language and then restricted to a given document for IR purposes [2]. LAs are identified by looking at pairs of words found in close proximity to each other. It has been described elsewhere [2] how LAs, when used as indexing units, improve precision of search by disambiguating terms. Juru's profiling component uses this technique as part of the profiling process to improve search precision by extracting lexical affinities from the text.

During query evaluation, the query profile is constructed to include the query's lexical affinities in addition to its individual terms. This is achieved by finding all pairs of words found close to each other in a window of some predefined small size (the sliding window is only defined within a sentence) based on the lexicographic evidence that 98% of LAs in the English language relate words that are in a distant of +/-5 words (see [2] for more details). For each *LA=(t1,t2),* Juru creates a pseudo posting list by merging the posting lists of *t1* and *t2*. It finds all documents in which these terms appear close to each other, and adds them to the posting list of the *LA* with all the relevant occurrence information. After creating the posting list, the new *LA* is treated by the retrieval algorithm as any other term in the query profile.

Lexical affinities can improve search results significantly, especially for short queries. The user can control whether to use LAs or not in the retrieval process. The user can also control the relative weight between keywords and *LA*s, thus giving more (or less) significance to *LAs* in relation to simple keywords in computing the relevance score. Figure 1 shows the relation between the system's precision and the relative weight given to *LAs* and to simple keywords. For a zero weight the queries are constructed with no *LAs*. For a weight of one the queries consist of *LAs* only (i.e., keywords are ignored). The experiments were done on the Wt10g collection with ad-hoc topics 501-550. Queries were formulated from the topic's title. From the graph we can see that the optimal relative weight is 0.3 for precision at 10 and 0.5 for avg. precision. For the experiments at TREC 10 described in this paper the LA weight was fixed to 0.3.

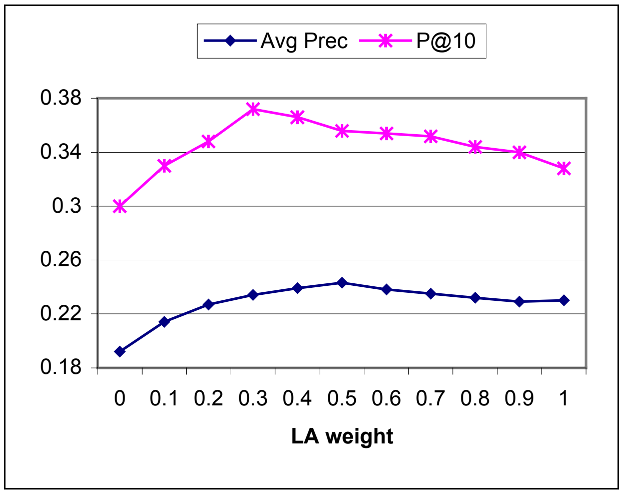

Figure 1: relation between the system's precision and the relative weight given to *LAs*

3 Index Pruning

Indexing a large collection of documents might result in extremely large index files that are difficult to maintain. Therefore, it is important to utilize efficient compression methods for index files. There are two complementary approaches: *lossless compression* and *lossy compression*. Lossless approaches do not lose any information; instead, they use more efficient data structures. Thus, under lossless approaches, posting lists have a very compact representation. On the other hand, under lossy approaches, certain information is discarded.

We propose here lossy methods that prune the index at the posting level. That is, in our approach, a term can be retained in the index, but some document postings may be eliminated from this term's posting list. The idea is to remove those postings whose potential contribution to the relevance score of a document is so small that their removal will have little effect on the accuracy of the system. The selection of which document postings to prune is guided by certain user-specified parameters.

Our goal is to perform index pruning in such a way that a human "cannot distinguish the difference" between the results of a search engine whose index is pruned and one whose index is not pruned. Therefore, as in any lossy compression technique, we wish to remove the least important terms from the index, so that the visible effects of the compression (in terms of the results obtained) are very small. Thus, the question we need to address is how to identify the least important entries in the index.

We begin with the usual assumption that for each query, there is a *scoring function* that assigns a score to each document, so that the documents with the highest scores are the most relevant. The scoring function is often based on a 2-dimensional scoring table, A, indexed by terms and documents. Table entries are set according to the scoring model of the search engine; thus, $A(t,d)$ is the score of document d for term t.

The first static pruning algorithm that we consider removes from the index all posting entries whose corresponding table values are bounded above by some fixed cutoff threshold. We refer to this type of pruning as *uniform pruning*, since the threshold is uniformly chosen, with the same cutoff value being used for every term. Uniform pruning has an obvious drawback. Low-scoring terms may have all of their entries in the index pruned away. Therefore, given a query consisting only of low-scoring terms, the pruned index may fail to provide any good results for this query.

This insight leads us to propose a second, and more sophisticated, pruning algorithm, in which the cutoff threshold may depend on the term. We refer to this type of pruning as *term-based pruning*. Term-based pruning guarantees that each term will have some representative postings left in the index. Therefore, queries with low-scoring terms will fare better than under uniform pruning. How do we determine the cutoff thresholds? We are guided by the intuition that all we really care about are the top k documents, since this is all the user sees. Thus, we care only about whether the pruned index returns the same top k documents; we do not care about the score it might assign to the remaining documents. Our term-based pruning algorithm attempts to minimize the effect of pruning on the top k results for each query.

Recall that the scoring table is not stored as such in the Juru index. Instead, each term is stored with an associated posting list. The following algorithm describes how to prune a given inverted file using the top k pruning algorithm. The algorithm takes as input an inverted file I, along with the parameters k and ε, and creates a pruned inverted file. Note that the entries of the scoring table A are computed on a term-by-term basis in order to find the cutoff value for each particular posting list.

```
Top k prune($I,k,\varepsilon$)
    For each term $t$ in $I$
        Retrieve the posting list $P_t$ from $I$
        If $|P_t| > k$
            For each entry $d$ in $P_t$
                Compute $A(t,d)$ according to the scoring model
                Let $z_t$ be the $k$th best entry in row $t$ of $A$
                $\tau_t = \varepsilon * z_t$
                For each entry $d$ in $P_t$
                    If $A(t,d) < \tau_t$ remove entry $d$ from $P_t$
        Save ($t$, $P_t$) in the pruned inverted file
```

The time complexity of the pruning algorithm is linearly proportional to the index size. For each term *t*, the algorithm first computes a threshold by finding the *k*th best entry in the posting list of *t* (this can be done in *O(N)* time, where *N* is the number of documents). It then scans the posting list to prune all the entries smaller than the threshold. Thus, if there are *M* terms in the index, the time complexity of the algorithm is *O(M*N)*.

In [1] we gave a formal proof that for all queries with a moderate number of search terms, the results obtained from the pruned index are indistinguishable from those obtained from the original index.

4 Experimental Results

Our experiments tested the impact of pruning on the search results. First we created a sequence of pruned indices using the uniform pruning algorithm where we varied τ, the cutoff threshold. Next we created a sequence of pruned indices by invoking the term-based pruning algorithm, where we fixed *k* to 10, and used varying values of ε. For each index we ran 50 queries, constructed automatically from the titles of topics 501-550. Our first experiment tested the effect of pruning on the similarity of the top results to the top results of the original index. The second tested the effect of pruning on the precision of the results.

4.1 The effect of pruning on similarity

The similarity of the top results was measured by two metrics. First, the *symetric difference* between the top 10 lists that evaluates the similarity between two lists by considering the common presence/absence of items in both lists. Second, a variation of *Kendall's tau* measure that considers not only the common presence/absence of items in the lists but also their rank. The symmetric difference is evaluated as follows: if *y* is the size of the union of the two top 10 lists, and *x* is the size of the symmetric difference, then we take the symmetric difference score to be $1 - x/y$. This score lies between 0 and 1. The highest score of 1 occurs precisely when both lists are identical (although the order of results may be different), and the lowest score of 0 occurs precisely when the two lists are disjoint.

The second metric we used is a variation of Kendall's tau method that was obtained in [8] and used in [1]. The original Kendall's tau method for comparing two permutations assigns a penalty for each pair of distinct items for which one item appears before the second in one permutation and the second appears before the first in the other permutation. The sum of the penalties over all pairs reflects the overall penalty. The modified version of Kendall's tau handles the case where we care about, comparing the top 10 in one list against the top 10 in another list, rather than comparing permutations. The penalties assigned for each pair of distinct items are redefined, since two distinct items might not appear in the top 10 of one or both lists. By normalizing the sum of penalties to lie between 0 and 1, the highest score of 1 occurs precisely when both lists are the same and in the same order, and the lowest score of 0 occurs precisely when the two lists are disjoint. More details appear in [1,8].

Figure 2 shows the similarity between the top 10 results of the original index and the pruned index, at varying levels of pruning, for both uniform pruning and term-based pruning. The relatively high similarity between the top 10 lists for moderate pruning levels supports the claim that the top 10 results of the pruned indices are very similar to the top 10 results of the original index. Surprisingly, in contrast to our previous results [10], there is no advantage for term-based pruning over uniform pruning. Both algorithms create pruned indices with similar behavior in terms result similarity.

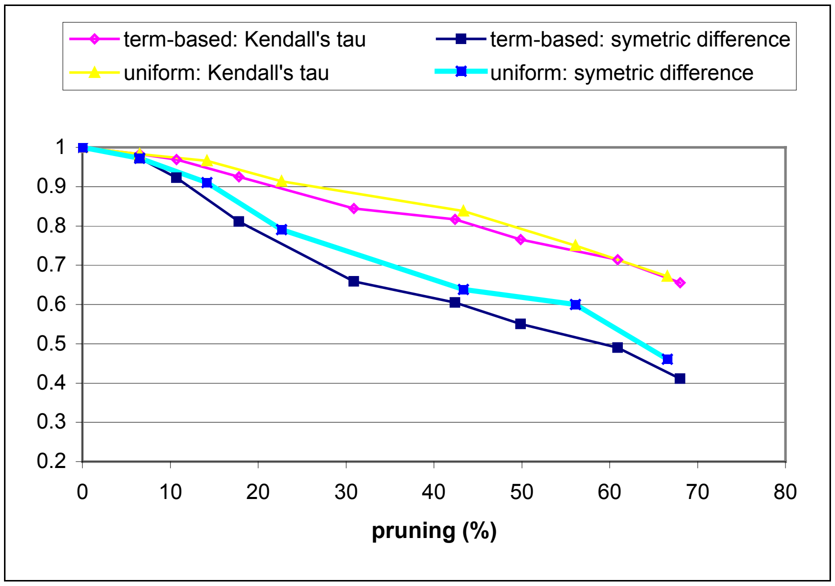

Figure 2: similarity as a function of pruning

4.2 The effect of pruning on precision

In order to measure the effect of pruning on precision we had to wait for TREC's official results. Four runs were submitted to TREC for evaluation. The first run consists of the results obtained from Juru's core index. The second and the third runs were obtained from two pruned indices created by the term-based pruning algorithm with parameters $k = 10$, $\varepsilon = 0.05$ (10.7% pruning) and $k = 10$, $\varepsilon = 0.1$ (17.8% pruning), respectively. The fourth run consists of the results of an experiment we performed with query expansion. Our expansion method failed to improve search performance, thus we ignore it in this report.

The following table shows the precision of the official runs submitted to TREC. The results support our claim that P@10 is barely affected for short queries even after significant pruning. Furthermore, while there is some loss in the mean average precision (MAP), it is negligible.

ε	Index size	Pruning (%)	MAP	P@10
0	3.53 GB	0	0.211	0.362
0.05	3.15 GB	10.7	0.207	0.362
0.1	2.9 GB	17.8	0.205	0.360

Figure 3 shows P@10 results obtained from the core index and the pruned indices for all the Web Track ad-hoc queries. For most of the queries (35 queries), P@10 remained the same. For 8 of the queries, it even improved and only 7 queries exhibited some loss in precision, where the largest loss is 0.2 (for query 530).

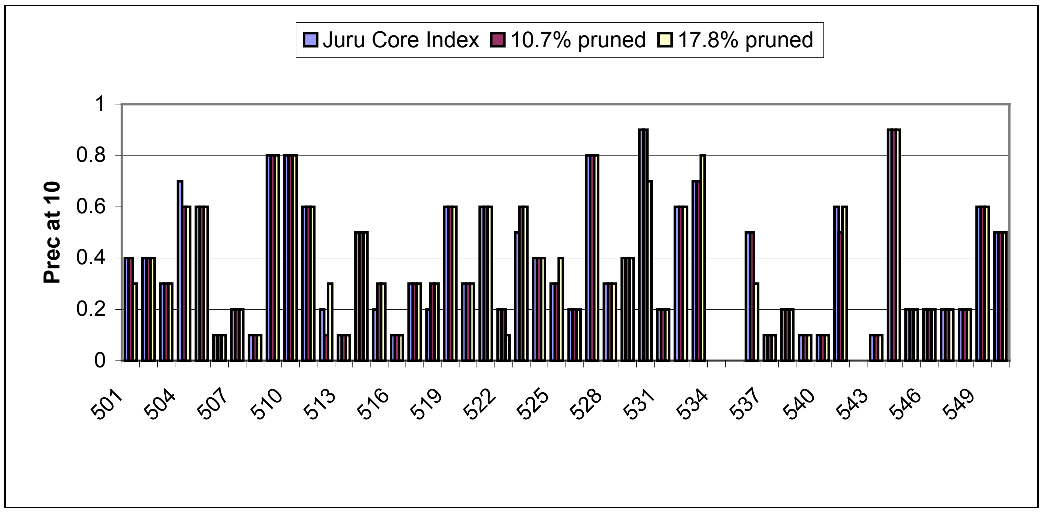

Figure 3: P@10 of topics 501-550 for the three official runs

Figure 4 shows the MAP for all queries obtained from the three runs. Although for many queries there is some decrease in precision, the drecrease is quite small (maximum of 16% loss for topic 504).

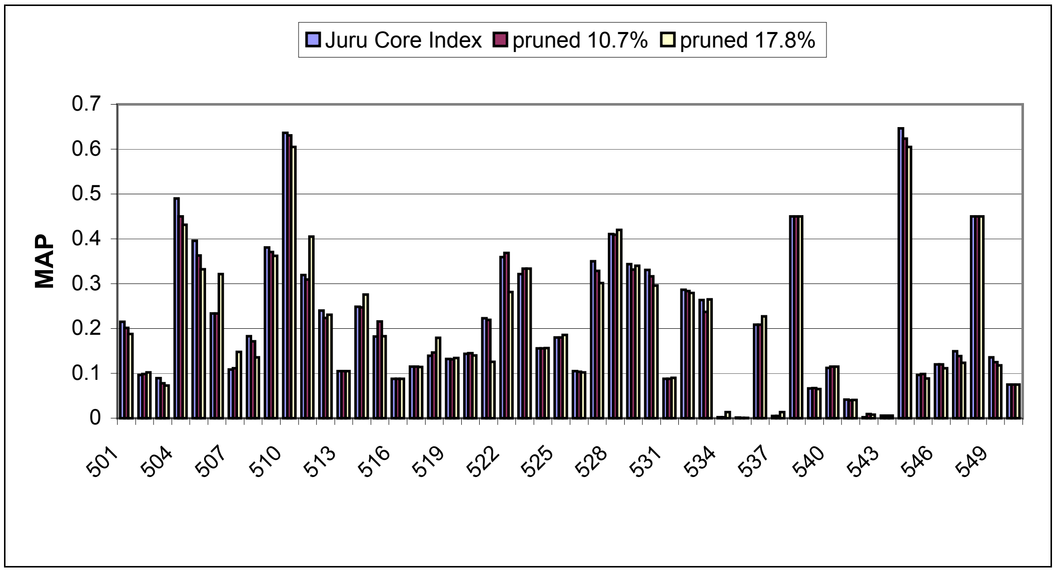

Figure 4: MAP of topics 501-550 of the three official runs

After receiving TREC's official results, we repeated the experiments with several additional indices created by our pruning algorithms. The goal was to test how much further we could prune the indices, before significant loss in precision occurs. We used the newly published "qrels" files to measure precision. Figure 5 shows the impact of pruning on precision as measured by MAP and P@10. From these tests, it is apparent that P@10 remains more or less stable up to 40% pruning. There is a slight decrease in average precision at 30% pruning, but a significant loss of MAP also occurs only at 40% pruning. As for the similarity experimental results, in contrast to our previous results [10], there is no advantage for term-based pruning over uniform pruning.

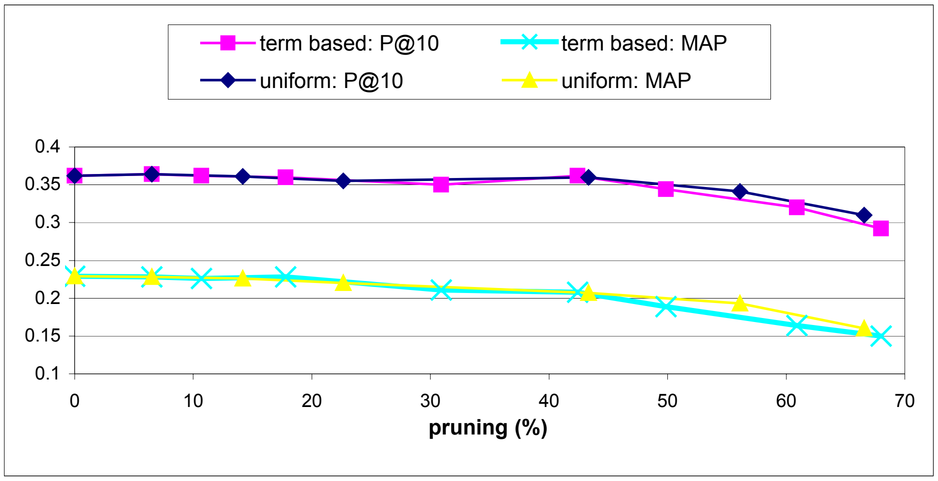

Figure 5: the impact of pruning on precision as measured by MAP and P@10.

5 . Summary

The results we obtained from the TREC experiments support and corroborate our previous results with index pruning: it is possible to discard a significant part of the index, and still attain answers that are almost as good as those obtained from the full index. By reducing the index size, it becomes feasible to index large text collections such as the Web track's data on low-end machines. Furthermore, using our method, Web search engines can significantly decrease the burden of storing extremely large indices by removing entries that have no influence on search results. Our experiments show that we can reduce the index size by 40% while attaining the same P@10 (which is most critical for Web search engines), and with only a slight decrease in the mean average precision.

In addition to validating the pruning methods, the Web Track results also demonstrate the overall high quality of the Juru search engine. From the initial results available at this point, it is apparent that Juru is significantly above the median of results attained by all participants for most of the queries as well as in overall precision.

References

[1] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici. Y. Maarek, A. Soffer. Static Index Pruning for Information Retrieval Systems. Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 43-50, New Orleans, 2001.
[2] Y. Maarek and F. Smadja. Full text indexing based on lexical relations: An application: Software libraries. Proceedings of the 12th International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 198-206, 1989.
[3] G. Grefenstette and P. Tapainen. What is a word? What is a sentence? Problems of Tokenization. In Proceedings of the 3rd International Conference on Computational Lexicography (COMPLEX '94), Research Institute for Linguistics, Hungarian Academy of Sciences, Budapest, 1994.
[4] M. F. Porter. An Algorithm for Suffix Stripping. Program 14(3), pages 130 – 137, 1980.
[5] I. H. Witten, A. Moffat, and T. C. Bell. Managing Gigabytes, Morgan Kaufman Publishers, San Francisco, California, 1999.
[6] C. Buckley and A. Singhal and M. Mitra and G. Salton. New retrieval approaches using SMART: TREC 4. Proceedings of the Fourth Text REtrieval Conference (TREC-4), pages 25 – 48, 1995
[7] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. Journal of the American Society of Information Science 41(8), pages 581 – 589, 1990.
[8] R. Fagin, R. Kumar, D. Sivakumar. Top *k* orderings and near metrics. To appear, 2002.