University of Otago at INEX 2010

Xiang-Fei Jia, David Alexander, Vaughn Wood and Andrew Trotman

Computer Science, University of Otago, Dunedin, New Zealand

Abstract. In this paper, we describe University of Otago's participation in Ad Hoc, Link-the-Wiki Tracks, Efficiency and Data Centric Tracks of INEX 2010. In the Link-the-Wiki Track, we show that the simpler relevance summation method works better for producing Best Entry Points (BEP). In the Ad Hoc Track, we discusses the effect of various stemming algorithms. In the Efficiency Track, we compare three query pruning algorithms and discusses other efficiency related issues. Finally in the Data Centric Track, we compare the BM25 and Divergence ranking functions.

1 Introduction

In INEX 2010, University of Otago participated in the Ah Hoc, Link-the-Wiki Tracks, the Efficiency and Data Centric Tracks. In the Link-the-Wiki Track, we talk about how our linking algorithm works using the Te Ara collection and the newly developed assessment tool. In the Ad Hoc Track, we show the performance of our stemming algorithm using Genetic Algorithms and how it performs against other stemming algorithms. In the Efficiency Track, we discusses the performance of our three pruning algorithms; The first is the original *topk* (originally described in INEX 2009), an improved version of the *topk* and the *heapk*. Finally in the Data Centric Track, we compare the BM25 and Divergence ranking functions.

In Section 2, related work is discussed. Section 3 explains how our search engine works. Section 4, 5, 6 and 7 talk about how we performed in the corresponding Tracks. The last section provides the conclusion and future work.

2 Related Work

2.1 The Link-the-Wiki Track

The aim of the INEX Link-the-Wiki track is to develop and evaluate link recommendation algorithms for large hypertext corpora.

Before 2009, Wikipedia was the only corpus used in the Link-the-Wiki track; the task was to link related Wikipedia documents to each other, with or without providing specific anchor locations in the source documents. In 2009, the *Te Ara Encyclopedia of New Zealand* was used alongside Wikipedia, and tasks included producing links within each of the two corpora, and linking articles in one corpus to articles in the other.

Work has been done on the topic of hypertext link recommendation by a number of people both within the INEX Link-the-Wiki track and outside of

it. It is difficult to compare INEX-assessed algorithms with non-INEX-assessed algorithms because the assessment methodology plays a large part in the results, so this section will focus on algorithms from within INEX.

For Wikipedia, the two most successful link-recommendation algorithms are due to Kelly Itakura [1] and Shlomo Geva [2].

Itakura's algorithm chooses anchors in a new document by calculating the probability (γ) that each phrase, if found in the already-linked part of the corpus, would be an anchor. If γ exceeds a certain threshold (which may be based on the length of the document), the phrase is used as an anchor. The target for the link is chosen to be the most common target for that anchor among existing links. The formula for γ for a given phrase P is:

$$\gamma = \frac{\text{number of occurrences of } P \text{ in the corpus as a link}}{\text{number of occurrences of } P \text{ in the corpus altogether}}$$

Geva's algorithm simply searches for occurrences of document titles in the text of the orphan document. If such an occurrence is found, it is used as an anchor. The target of the link is the document whose title was found.

2.2 The Ad Hoc Track

In the Ad Hoc Track, we compare the performance of our stemming algorithm using Genetic Algorithms with other stemming algorithms.

The S Stripper consists of three rules. These rules are given in Table 1. It uses only the first matching rule. It has improved MAP on previous INEX Ad Hoc collections, from 2006-2009. This serves as a baseline for stemmer performance, and is an example of a weak stemmer (It does not conflate many terms).

ies	→ y
es	→
s	→

Table 1: S Stripper rules. The first suffix matched on the left is replaced by the suffix on the right.

The Porter stemmer [3] has improved some runs for our search engine on previous INEX collections. It serves as an example of a strong stemmer. We use it here as a baseline for comparing stemmer performance.

People have found ways to learn to expand queries using thesauruses generation or statistical methods. Jones [4] used clustering methods for query expansion. We have been unable to find any mention of symbolic learning used for stemming.

A similar method for improving stemming by using term similarity information from the corpus was used by Xu and Croft [5]. Their work uses the Expected

Mutual Information Measure. Instead we have used Pointwise Mutual Information and the Jaccard Index. These were chosen as the best out of a larger group of measures.

2.3 The Efficiency Track

The following discussion of the related work is taken from our published paper in ADCS 2010 [6].

Disk I/O involves reading query terms from a dictionary (a vocabulary of all terms in the collection) and the corresponding postings lists for the terms. The dictionary has a small size and can be loaded into memory at start-up. However, due to their large size, postings are usually compressed and stored on disk. A number of compression algorithms have been developed and compared [7,8]. Another way of reducing disk I/O is caching, either at application level or system level [9,10]. Since the advent of 64-bit machines with vast amounts of memory, it has become feasible to load both the dictionary and the compressed postings into main memory, thus eliminating all disk I/O. Reading both dictionary and postings lists into memory is the approach taken in our search engine.

The processing (decompression and similarity ranking) of postings and subsequent sorting of accumulators can be computationally expensive, especially when queries contain frequent terms. Processing of these frequent terms not only takes time, but also has little impact on the final ranking results. Postings pruning at query time is a method to eliminate unnecessary processing of postings and thus reduce the number of non-zero accumulators to be sorted. A number of pruning methods have been developed and proved to be efficient and effective [11,12,13,14,15,16]. In our previous work [16], the *topk* pruning algorithm partially sorts the static array of accumulators using an optimised version of quick sort [17] and statically prunes postings. In this paper, we present an improved *topk* pruning algorithm and an new pruning algorithm based on heap data structure.

Traditionally, term postings are stored in pairs of <document number, term frequency> pairs. However, postings should be impact ordered so that most important postings can be processed first and the less important ones can be pruned using pruning methods [18,14,15]. One approach is to store postings in order of term frequency and documents with the same term frequency are grouped together [18,14]. Each group stores the term frequency at the beginning of the group followed by the compressed differences of the document numbers. The format of a postings list for a term is a list of the groups in descending order of term frequencies. Another approach is to pre-compute similarity values and use these pre-computed impact values to group documents instead of term frequencies [15]. Pre-computed impact values are positive real numbers. In order to better compress these numbers, they are quantised into whole numbers [19,15]. Three forms of quantisation method have been proposed (*Left.Geom*, *Uniform.Geom*, *Right.Geom*) and each of the methods can better preserve certain range of the original numbers [15]. In our search engine, we use pre-computed

BM25 impact values to group documents and the differences of document numbers in each group are compressed using Variable Byte Coding by default. We choose to use the *Uniform.Geom* quantisation method for transformation of the impact values, because the *Uniform.Geom* quantisation method preserves the original distribution of the numbers, thus no decoding is required at query time. Each impact value is quantised into an 8-bit whole number.

Since only partial postings are processed in query pruning, there is no need to decompress the whole postings lists. Skipping [12] and blocking [20] allow pseudo-random access into encoded postings lists and only decompress the needed parts. Further research work [21,22] represent postings in fixed number of bits, thus allowing full random access. Our search engine partially decompress postings list based on the worst case of the static pruning. Since we know the parameter value of the static pruning and the biggest size of an uncompressed impact value (1 byte), we can add these together to find the cut point for decompression. We can simply hold decompression after that number of postings have been decompressed.

3 System Overview

3.1 Indexer

Memory management is a challenge for fast indexing. Efficient management of memory can substantially reduce indexing time. Our search engine has a memory management layer above the operating system. The layer pre-allocates large chunks of memory. When the search engine requires memory, the requests are served from the pre-allocated pool, instead of calling system memory allocation functions. The sacrifice is that some portion of pre-allocated memory might be wasted. The memory layer is used both in indexing and in query evaluation. As shown previously in [16], only a very small portion of memory is actually wasted.

The indexer uses hashing with a collision binary tree for maintaining terms. We tried several hashing functions including Hsieh's super fast hashing function. By default, the indexer uses a very simple hashing function, which only hashes the first four characters of a term and its length by referencing a pre-defined look-up table. A simple hashing function has less computational cost, but causes more collisions. Collisions are handled by a simple unbalanced binary tree. We will examine the advantages of various hashing and chaining algorithms in future work.

Postings lists can vary substantially in length. The indexer uses various sizes of memory blocks chained together. The initial block size is 8 bytes and the re-size factor is 1.5 for the subsequent blocks.

The indexer supports either storing term frequencies or pre-computed impact values. A modified BM25 is used for pre-computing the impact values. This variant does not result in negative IDF values and is defined thus:

$$RSV_d = \sum_{t \in q} log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1+1)\, tf_{td}}{k_1\left((1-b)+b\times\left(\frac{L_d}{L_{avg}}\right)\right)+tf_{td}}$$

here, N is the total number of documents, and df_t and tf_{td} are the number of documents containing the term t and the frequency of the term in document d, and L_d and L_{avg} are the length of document d and the average length of all documents. The empirical parameters k_1 and b have been set to 0.9 and 0.4 respectively by training on the previous INEX Wikipedia collection.

In order to reduce the size of the inverted file, we always use 1 byte to store term frequencies and pre-computed impact values. This limits to a maximum value of 255. Term frequencies which have values larger than 255 are simply truncated. Truncating term frequencies could have an impact on long documents. But we assume long documents are rare in a collection and terms with high frequencies in a document are more likely to be common words. Pre-computed impact values are transformed using the *Uniform.Geom* quantisation method.

As shown in Figure 1, the index file has five levels of structure. In the top level, original documents in compressed format can be stored. Storing original documents is optional, but is required for focused retrieval.

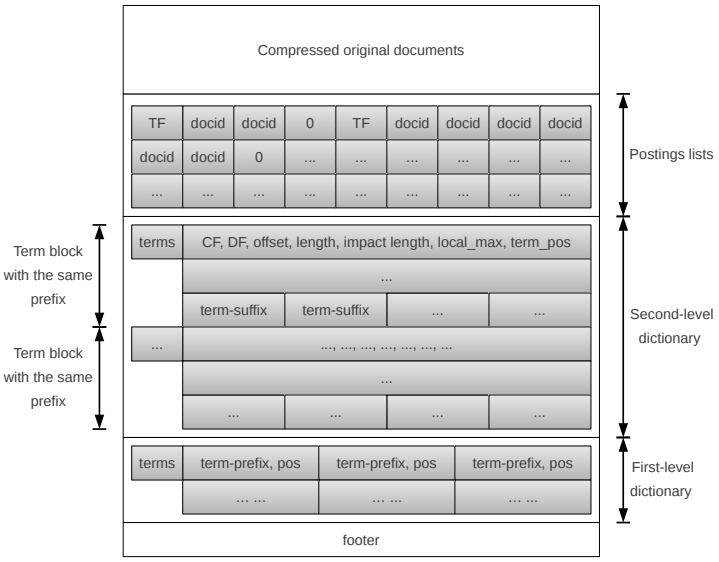

Fig. 1: The index structures.

Instead of using the pair of <document number, term frequency> for postings, we group documents with the same term frequency (or the impact value) together and store the term frequency (or the impact value) at the beginning of each group. By grouping and impacting order documents according to term frequencies (or impact values), during query evaluation we can easily process documents with potential high impacts first and prune the less important doc-

uments at the end of the postings list. The difference of document ids in each group are then stored in increasing order and each group ends with a zero. Postings are compressed with Variable Byte coding.

The dictionary of terms is split into two parts. Terms with the same prefix are grouped together in a term block. The common prefix (only the first four characters) is stored in the first level of the dictionary and the remaining are stored in the term block in the second level. The number of terms in the block is stored at the beginning of the block. The term block also stores the statistics for the terms, including collection frequency, document frequency, offset to locate the postings list, the length of the postings list stored on disk, the uncompressed length of the postings list, and the position to locate the term suffix which is stored at the end of the term block.

At the very end of the index file, the small footer stores the location of the first level dictionary and other values for the management of the index.

3.2 Query Evaluation

At start-up, only the the first-level dictionary is loaded into memory by default. To process a query term, two disk reads have to be issued; The first reads the second-level dictionary. Then the offset in that structure is used to locate postings. The search engine also supports a command line option which allows loading the whole index into memory, thus totally eliminating I/O at query time.

An array is used to store the accumulators. We used fixed point arithmetic on the accumulators because it is faster than the floating point.

For last year INEX, we developed the *topk* algorithm for fast sorting of the accumulators. It uses a special version of quick sort [17] which partially sorts the accumulators. A command line option (lower-k) is used to specify how many top documents to return.

Instead of explicit sorting of all the accumulators, we have developed an improved version of *topk*. During query evaluation, it keeps track of the current top documents and the minimum partial similarity score among the top documents. The improved *topk* uses an array of pointers to keep track of top documents. Two operations are required to maintain the top documents, i.e. *update* and *insert*. If a document is in the top documents and gets updated to a new score, the improved *topk* simply does nothing. If a document is not in the top k and gets updated to a new score which is larger than the minimum score, the document needs to be inserted into the *topk*. The insert operation is accomplished by two linear scans of the array of pointers; (1) the first scan locates the document which has the minimum score and swap the minimum document with the newly updated document, (2) the second finds the current minimum similarity score.

Based on the *topk* algorithm, we have further developed a new algorithm called *heapk*. It uses a minimum heap to keep track of the top documents. Instead of using the minimum similarity score, *heapk* uses bit strings to define if a document is among the top k. The heap structure is only built once which is when the number of top slots are fully filled. If a document is in the heap and gets updated to a new score, *heapk* first linearly scans the array to locate the

document in the heap and then partially updates the structure. If a document is not in the heap and the newly updated score is larger than the minimum score (the first pointer) in the heap, *heapk* partially inserts the document into the heap.

The upper-K command line option is used for static pruning of postings. It specifies a value, which is the number of postings to be processed. Since only part of the postings lists is processed, there is no need to decompress the whole list. Our search engine partially decompress postings lists based on the worst cast. Since we know the parameter value of upper-K and the biggest size of an uncompressed impact value (1 byte), we can add these together to find the cut point for decompression.

4 The Link-The-Wiki Track

In this year's Link-the-Wiki track, the only corpus used was the *Te Ara Encyclopedia of New Zealand*. Wikipedia was abandoned as a corpus because it had become too easy for algorithms to score highly according to the metrics used by INEX. This is believed to be because of characteristics of Wikipedia that Te Ara does not possess. Te Ara is therefore of interest because it presents challenges that Wikipedia does not.

It is also of interest because its maintainers (New Zealand's Ministry of Culture and Heritage) have asked for links to be incorporated into the official, public version of their encyclopedia. This is an opportunity for these linking algorithms to be tested in a real-world application.

Our participation in the Link-the-Wiki track is detailed in the rest of this section. First, the differences between Wikipedia and Te Ara are outlined, as well as the possible ways to develop linking algorithms for Te Ara. Then, our own linking algorithm is explained, and its assessment results given. Finally, our contribution to the Link-the-Wiki assessment process is explained.

4.1 Differences between Wikipedia and Te Ara

The most important difference between Wikipedia and Te Ara is that Te Ara has no existing links. The Link-the-Wiki Track has always been to take a single "orphan" (a document whose incoming and outgoing links have been removed) and produce appropriate links to and from it, using the remainder of the corpus (including any links that do not involve the orphan) as input if desired. This meant that algorithms could statistically analyse the anchors and targets of the existing links in the corpus, using that information to decide what kind of links would be appropriate for the orphan document. Itakura's algorithm (described in Section 2) is an example of one that does so, and it has been consistently successful on Wikipedia.

In Te Ara this is not possible. The problem is not merely the lack of links, but that the encyclopedia was not written with links in mind. In any body of writing there are a number of different ways to refer to a given topic, but in a

hypertext corpus such as Wikipedia, writers tend to use existing article titles as "canonical names" to refer to the topics of those articles. The absence of this in Te Ara renders an approach such as Geva's algorithm less effective.

Wikipedia and Te Ara are also organised in very different ways. Te Ara is primarily a record of New Zealand history, and the discussion of any given topic may be spread among several articles, each of which may discuss other topics as well. This is especially true of topics that are relevant to both the indigenous and colonial inhabitants of New Zealand; and also topics that have been relevant over a long period of time. In Wikipedia, even such wide-ranging topics are typically centred around a single article.

4.2 Adapting to the differences in Te Ara

Without the possibility of using previous years' best-performing algorithms directly on Te Ara, we were left with two options: we could either find a way to "map" Wikipedia documents to their closest Te Ara counterparts, and then translate Wikipedia links into Te Ara links; or we could devise a new linking algorithm that did not rely on existing links at all.

We chose the latter option because, as discussed above, Te Ara is organised very differently from Wikipedia, and finding a suitable mapping would have been difficult. The algorithm we used is described below.

4.3 Algorithm

The main premise behind our linking algorithm is that Te Ara documents are less "to-the-point" than Wikipedia documents (that is, a single Te Ara article tends to touch on numerous related topics in order to "tell a story" of some sort), and therefore it is important to take into account the immediate context of a candidate anchor or entry-point, as well as the more general content of the two documents being linked.

Three sets of files were created and indexed using our search engine (described in Section 3). In the first, each document was contained within a separate file. In the second, each section of each document was contained within a separate file. The third was the same, but only included the section headings rather than the body text of each section. In this way, we were able to vary the level of target-document context that was taken into account when searching for possible entry-points for a given link.

Within each source document, candidate anchors were generated. Every maximal sequence of consecutive words containing no stopwords or punctuation marks was considered as a candidate anchor. The purpose of this was to avoid using large portions of sentences as anchors merely because all the words appear in the target document.

For each candidate anchor, various levels of context around the anchor (document, paragraph, sentence, and clause) were extracted from the source document. Each anchor context, as well as the anchor text itself, was used to query

for possible targets against whichever one of the three target file-sets provided the level of context closest in size to the source context. If a particular document (or section) appeared in the query results for the anchor text itself, and for at least one of the chosen contexts, it was used as a target for that anchor. The target was given a relevance score, which was a weighted average of the relevance scores given by BM25 for each of the different contexts' queries, based on our estimate of their importance.

24 runs were produced by varying the following 4 parameters:

- *Full-document anchor context* Whether or not the entire source document of an anchor was used as one of its contexts. If not, the largest level of context was the paragraph containing the anchor.
- *Relevance summation method* How the total relevance score for a link was added up. In one method, the relevance scores for a target, queried from all levels of context and from the anchor itself, were simply averaged using the predetermined weights. In the other method, the values averaged were the squared differences between the relevance scores for each context and from the anchor. The rationale for the second method was that if a target was much more relevant to the anchor context than the anchor, then a nearby anchor would probably be better than the current one.
- *Relevance score contribution* Whether all of the weights for the anchor contexts were non-zero, or just the weight for the largest context. When a context has a weight of zero, it still contributes to the choice of targets for an anchor, but not to their scores.
- *Target contexts* Which target contexts the anchor texts themselves were directly queried against (headings, sections or both).

4.4 Results

This section details the results of assessing the 24 runs described in Section 4.3.

Table 2 shows the mean average precisions for the BEPs produced by each run. Precision/Recall graphs are included in the track overview paper.

All the full document contexts runs outperformed the paragraph context runs. This result suggests that context is important when predicting links for Te Ara, and a generalisation of the result that context matters in Focused Retrieval in general.

The difference squared method for summation always worked better than the sum method, and the single relevance context worked best (in that order). This suggests that although context is important in identifying links, the best link to use is determined by using just one context.

The best target context to use is the heading, followed by heading and section, then just section. This results suggests that headings are important for identifying targets – something that was show to be the case with the Wikipedia link-the-wiki.

Run	Context	Summation	Contribution	Element	MAP
1	Article	Diff	Single	Heading	0.0906
2	Article	Diff	Single	Both	0.0906
3	Article	Diff	Single	Section	0.0868
4	Article	Diff	Average	Heading	0.0868
5	Article	Diff	Average	Both	0.0863
6	Article	Diff	Average	Section	0.0768
7	Article	Sum	Single	Heading	0.0767
8	Article	Sum	Single	Both	0.0703
9	Article	Sum	Single	Section	0.0700
10	Article	Sum	Average	Heading	0.0481
11	Article	Sum	Average	Both	0.0481
12	Article	Sum	Average	Section	0.0136
13	Paragraph	Diff	Single	Heading	0.0136
14	Paragraph	Diff	Single	Both	0.0102
15	Paragraph	Diff	Single	Section	0.0102
16	Paragraph	Diff	Average	Heading	0.0102
17	Paragraph	Diff	Average	Both	0.0102
18	Paragraph	Sum	Average	Section	0.0102
19	Paragraph	Sum	Single	Heading	0.0102
20	Paragraph	Sum	Single	Both	0.0102
21	Paragraph	Sum	Single	Section	0.0102
22	Paragraph	Sum	Average	Heading	0.0102
23	Paragraph	Sum	Average	Both	0.0102
24	Paragraph	Sum	Average	Section	0.0102

Fig. 2: Results of the Otago runs in INEX 2010 Link-the-Wiki.

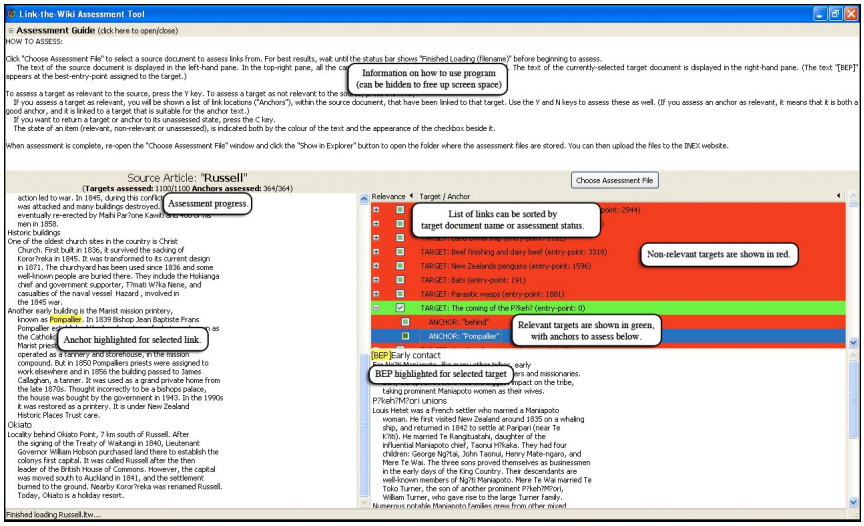

Fig. 3: An annotated screenshot of the 2010 assessment tool.

4.5 Assessment Tool

Apart from submitting runs to Link-the-Wiki, we also took over the task of maintaining the assessment tool.

Improvements have been made to the assessment tool every year. However, it is crucial to the quality of our results that the manual assessment process is made as easy as possible — it is difficult for assessors to produce reliable results if they cannot understand what they are being asked, if they do not have readily available all the information that they need to make an assessment, if they need to perform unnecessarily repetitive tasks to make assessments, or if the tool responds too slowly. Therefore, we decided to make further improvements.

We rewrote the assessment tool from scratch in C++ using the cross-platform GUI library GTK+, with SQLite databases for storing assessment information. This has resulted in a tool that responds to the user's requests quickly, even for large documents containing many links to be assessed.

We also made some changes to the layout of the GUI. The previous GUI only showed information about one target document at a time, whereas the new one shows a list of the titles of all target documents to be assessed, and shows the contents of the selected target document. Rather than having every link assessed, as was done previously, we only ask the assessor to assess links whose BEPs they have deemed relevant (the assumption being that an anchor cannot be relevant if its BEP is not). Figure 3 shows a screenshot of the new GUI.

As well as improving the quality of assessments in 2010, we hope that our changes to the assessment tool will reveal further areas for improvement in 2011. Our assessment tool collects usage statistics, the analysis of which should help us improve the tool.

Even before analysing these statistics we have been able to identify one possible area for improvement. It became clear while doing the assessment that the process would have been greatly sped up if "hints" had been provided to the assessor about whether a target was likely to be relevant. As the assessment for a particular topic progressed, the assessor could build up a list of "relevant" and "non-relevant" words for that topic, which would be highlighted whenever they appeared in a candidate target document, just as the Ah Hoc tool does. The assessor could ignore this if necessary, but it would help in many cases. However, it would be very important to use such a feature carefully so as not to bias the assessment process.

5 The Ad Hoc Track

5.1 Learning Stemmers

We previously learnt suffix rewriting stemmers using Genetic Algorithms. The stemmer referred to as the Otago Stemmer is one created part way through this work. Here we use it to address one problem with using assessments to learn recall enhancing methods like stemming. Pooled collections rely on the result lists of the participants to restrict the list of documents to assess. When we later

try to learn a recall enhancing method, finding documents which were not found by any participant cannot be rewarded by increases in Mean Average Precision. The goal of submitting runs with the Otago stemmer is to compare performance with the baselines where we can add documents to the pool.

Measure	Match this	Replace with this
0	shi	
2	ej	
4	ngen	
1	i	dops
4	nes	sy
0	ics	e
0	ii	sr
0	ito	ng
4	rs	tie
0	q	
4	al	
3	in	ar
0	ice	s
3	ic	
4	rs	tie
1	s	
1	f	uow
0	f	uow
0	q	
1	s	
2	que	sy
0	sl	anu
2	e	
1	f	
3	ague	dz
0	ean	

Table 2: The Otago Stemmer. Rule sections are separated by lines.

The rules of the Otago stemmer are shown in Table 2. Each rule of the stemmer uses a measure condition to ensure the length of the word is sufficient for a suffix to exist. This is taken from the Porter stemmer, and is an attempt to count the number of syllables. The measure of the word must be greater than or equal to the value for the rule. As an efficiency measure, any word to be stemmed must be longer than 3 characters. It also partitions the rules into sections. Only the first successful rule in a section is used. This was learnt on the INEX 2008 Wikipedia collection.

5.2 Refining Stemmers

We sought to improve the sets of terms that stemmers conflate. Additional terms found by the stemmer are only conflated if they are similar enough to the query term. We found a threshold value for several measures using an adaptive grid search on the INEX 2008 Wikipedia collection. Pointwise Mutual Information (PMI) and the Jaccard Index were found to aid performance, and we submitted runs using them to improve the Otago stemmer.

For both measures we used the term occurrences in documents as the probability distributions or sets to compare. For PMI, a threshold of 1.43 was found to give the best improvement. Only terms with similarity scores greater or equal to this were conflated. The PMI for two distributions x and y:

$$PMI(x, y) = log \frac{P(x, y)}{P(x)P(y)}$$

The Jaccard Index used a parameter of 0.00023 and is given between two sets of documents A and B by:

$$J(A, B) = \frac{A \cap B}{A \cup B}$$

5.3 Experimental Results

For the INEX 2010 Ad Hoc track we submitted 7 runs. Their performance is given in Table 3. These runs are combinations of stemmers and stemmer refinement.

The best run uses just the S Stripper. We find the Otago stemmer provides decent performance, and Porter to hurt performance a lot. Our baseline of no stemming occurs between the Otago and Porter stemmers.

Using PMI to improve the Otago stemmer proved successful. The Jaccard index on the same was less so. On the S stripper the Jaccard Index was found to harm performance excessively.

We forgot to submit one run, the PMI used on the S stripper. This run has been performed locally and gives a slight decrease in performance to just using the S stripper.

Rank	MAP	Run Name	Features
47	0.3012	v_sstem	S Stripper
54	0.2935	v_otago_w_pmi	Otago Stemmer with PMI refinement
58	0.2898	v_ostem_w_jts	Otago Stemmer with Jaccard Index refinement
59	0.2894	v_otago_stem_1	Otago Stemmer
61	0.2789	v_no_stem	No Stemming
74	0.2556	v_porter	Porter Stemmer
105	0.1102	v_sstem_w_jts	S Stripper with Jaccard Index refinement

Table 3: Stemming runs.

6 The Efficiency Track

6.1 Experiments

We conducted our experiments on a system with dual quad-core Intel Xeon E5410 2.3 GHz, DDR2 PC5300 8 GB main memory, Seagate 7200 RPM 500 GB hard drive, and running Linux with kernel version 2.6.30.

We conducted three sets of experiments, one for each of the *topk*, improved *topk,* and *heapk* algorithms. For the sets of experiments on the original *topk*, we used the same settings as our experiments conduced in INEX 2009. We want to compare the performance of the original *topk* with our improved *topk* and *heapk* algorithms.

The collection used in the INEX 2010 Efficiency Track is the INEX 2009 Wikipedia collection [23].

Collection Size	50.7 GB
Documents	2666190
Avg Document Length	880 words
Unique Words	11437080
Total Worlds	2347132312
Postings Size	1.2 GB
Dictionary Size	399 MB

(a)

Collection Size	50.7 GB
Documents	2666190
Avg Document Length	880 words
Unique Words	11186163
Total Worlds	2347132312
Postings Size	1.5 GB
Dictionary Size	390 MB

(b)

Table 4: (a) Summary of INEX 2009 Wikipedia Collection using term frequencies as impact values and without stemming. (b) Summary of INEX 2009 Wikipedia Collection using pre-computed BM25 as impact values and S-Striping for stemming.

The collection was indexed twice, one for the original *topk* and one for improved *topk* and *heapk*. For the original *topk*, term frequencies were used as impact values, no words were stopped and stemming was not used. For the improved *topk* and *heapk*, pre-computed BM25 similarity scores were used as impact values and S-String stemming was used. Table 4a and 4b show the summary of the document collection and statistics for the index file.

The Efficiency Track used 107 topics in INEX Ad Hoc 2010. Only title was used for each topic. All topics allow *focused*, *thorough* and *article* query evaluations. For the Efficiency Track, we only evaluated the topics for *article* Content-Only. During query evaluation, the terms for each topic were sorted in order of the maximum impact values of the terms.

For the sets of experiments on the improved *topk* and *heapk*, the whole index was loaded into memory, thus no I/O was involved at query evaluation time. For the original *topk*, only first-level dictionary was loaded into memory at start-up.

For the three sets of experiments, we specified lower-k parameter with k = 15, 150 and 1500 as required by the Efficiency Track. For each iteration of the

lower-k, we specified the upper-K of 10, 100, 1 000, 10 000, 100 000, 1 000 000. In total we submitted 54 runs. The lists of run IDs and the associated lower-k and upper-K values are shown in Table 5. Officially we submitted the wrong runs for the *heapk*. The runs has been corrected and are used in this paper and the MAiP measures are generated using the official assessment tools.

Lower-k	Upper-K	Original Topk	Improved Topk	Heapk
15	10	09topk-1	10topk-1	10heapk-1
15	100	09topk-2	10topk-2	10heapk-2
15	1000	09topk-3	10topk-3	10heapk-3
15	10000	09topk-4	10topk-4	10heapk-4
15	100000	09topk-5	10topk-5	10heapk-5
15	1000000	09topk-6	10topk-6	10heapk-6
150	10	09topk-7	10topk-7	10heapk-7
150	100	09topk-8	10topk-8	10heapk-8
150	1000	09topk-9	10topk-9	10heapk-9
150	10000	09topk-10	10topk-10	10heapk-10
150	100000	09topk-11	10topk-11	10heapk-11
150	1000000	09topk-12	10topk-12	10heapk-12
1500	10	09topk-13	10topk-13	10heapk-13
1500	100	09topk-14	10topk-14	10heapk-14
1500	1000	09topk-15	10topk-15	10heapk-15
1500	10000	09topk-16	10topk-16	10heapk-16
1500	100000	09topk-17	10topk-17	10heapk-17
1500	1000000	09topk-18	10topk-18	10heapk-18

Table 5: The lists of run IDs and the associated lower-k and upper-K values.

6.2 Results

This section talks about the evaluation and performance of our three sets of the runs, obtained from the official Efficiency Track (except for the *heapk*).

Figure 4 shows the MAiP measures for the original *topk*, improved *topk* and *heapk*. When upper-K has values of 150 and 1500, MAiP measures are much better than the upper-K 15. In terms of lower-k, MAiP measures approach constant at a value of 10 000. The best runs are 09topk-18 with a value of 0.2151, 10topk-18 with a value of 0.2304 and 10heapk-18 with a value of 0.2267 for the three algorithms respectively.

The MAiP measures are about the same for the improved *topk* and *heapk*. The subtle differences are when documents have the same similarity scores and the order of these documents can be different between the improved *topk* and *heapk*.

The MAiP measures of the original *topk* are quite different from the other two algorithms. Using term frequencies as impact values have better MAiP measures

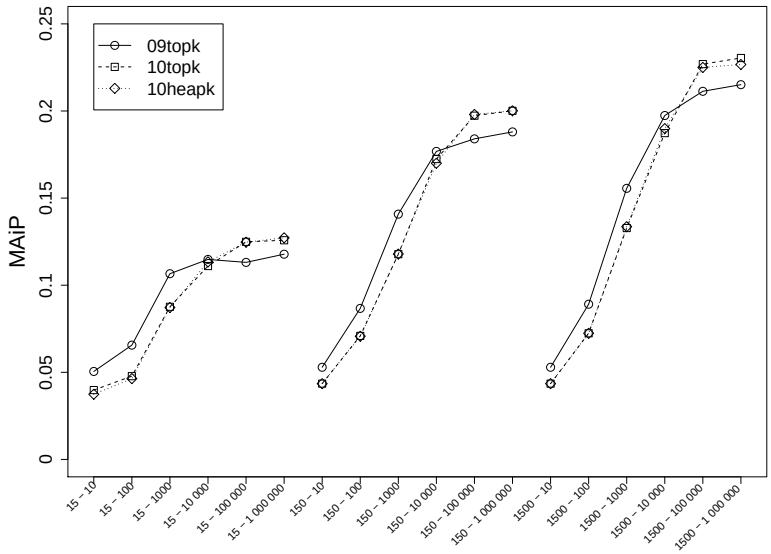

Fig. 4: MAiP measures for the original *topk*, improved *topk* and *heapk*.

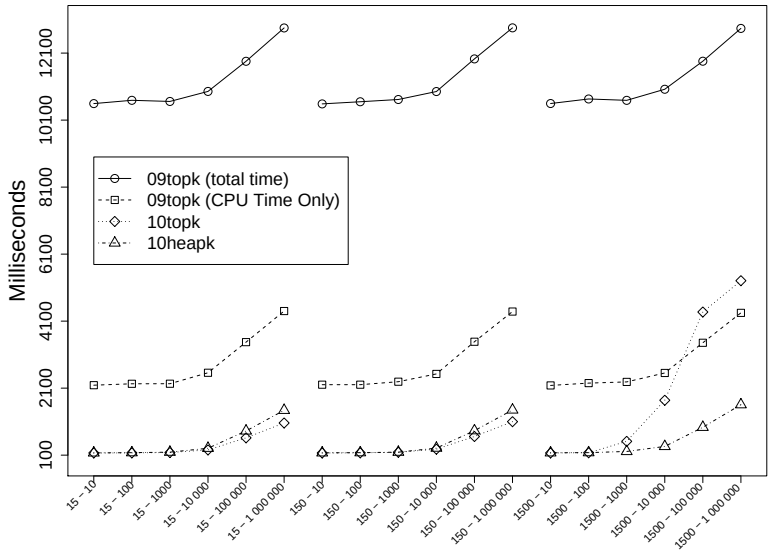

Fig. 5: Efficiency comparison.

when the values of lower-k and upper-K are small while pre-computed BM25 impact values have better MAiP measures when upper-K has a value larger than 10 000.

To have a better picture of the time cost for the three sets of the runs, we plotted the total evaluation times (including both CPU and I/O times) of all runs in Figure 5. The total times of both the improved *topk* and *heapk* are simply the CPU costs since the index was load into memory.

For the original *topk*, the total times were dominated by the I/O times. Regardless of the values used for lower-k and upper-K, the same number of postings were retrieved from disk, thus causing all runs to have the same amount of disk I/O.

We also plotted the CPU times of the original *topk* since we want to compare it with the other algorithms in terms of CPU cost. The differences of the CPU times between the original *topk* and the other two algorithms are the times taken for decompression of the postings lists and sorting of the accumulators. First, partial decompression was used in improved *topk* and *heapk* while the original *topk* did not. Second, the original *topk* used a special version of quick sort to partially sort all accumulators while the improved *topk* and *heapk* only keep track of the top documents only the final top documents got sorted.

For the original *topk*, the value of lower-k has no effect on the CPU cost, and values of 10 000 or above for upper-K causes more CPU usage.

For the improved *topk*, it performs the best when lower-k has a value of 15 and 150. However, for the set of the runs where the value of lower-k is 1500, the performance of the improved *topk* grows exponentially. This is caused by the linearly scans of the array of pointers to insert a new document into the top k.

For the runs when lower-k has a value of 15 and 150, the *heapk* has a small overhead compared with the improved *topk*, especially when upper-K has a large value. Well, the *heapk* performs the best when both lower-k and upper-K have large values.

7 The Data Centric Track

The collection used in the INEX 2010 Efficiency Track is the 2010 IMDB collection. The collection was indexed twice. The first index used pre-computed BM25 similarity scores as the impact values and the second used pre-computer Divergence similarity scores [24] as the impact values. For both indexes, no words were stopped and S-String stemming was used. Table 6 shows the results. With the ranking shown as (position / total runs), the results suggest that BM25 is a better ranking function than Divergence from Randomness for this collection, it consistently performed better regardless of the measure. They also suggest that BM25 whole document ranking is effective with our best run consistently in the top 6 regardless of how it is measured. We believe that, as is already the case in the ad hoc track, BM25 document ranking should be used as a baseline in future years in the document centric track.

Run ID	MAgP	MAiP	MAP
DC-BM25	0.2491 (#1/14)	0.1550 (#6/29)	0.3397 (#5/29)
DC-DIVERGENCE	0.1561 (#5/14)	0.1011 (#3/29)	0.2103 (#14/29)

Table 6: Effectiveness measure for the Data Centric Track

8 Conclusion and Future Work

8.1 The Link-the-Wiki Track

We have generated a number of runs for Te Ara. Given the inapplicability of Itakura and Geva's algorithms to Te Ara (see Section 4.1), we believe that this year's results are a step in the right direction towards a successful solution of what is still an unsolved problem: link recommendation in a corpus that has no existing links.

8.2 The Ad Hoc Track

We find that the S stripper is hard to beat. However it is possible to use machine learning to create a good stemmer. Furthermore such stemmers seem amenable to improvement using collection statistics. Of those PMI is a good measure to use. It was also found to be the best locally. This also confirms previous findings that Porter can have a variable effect on performance. Improvement using term similarity can also harm performance. We had seen this before when finding the parameters to use, so perhaps that might have been the consequence for the Jaccard Index. Of course, this refinement can only prevent terms from being stemmed together, so using it on such a weak stemmer would be expected to not do so well.

8.3 The Efficiency Track

We compared three of our query pruning algorithms. The original *topk* uses a special version of quick sort to sort all accumulators and return the top k documents. Instead of explicitly sorting all accumulators, the improved *topk* keeps tracks of the current top k documents and finally the top k documents are sorted and returned. Based on the improved *topk*, we have developed *heapk* which essentially is a minimum heap structure. The *heapk* algorithm has small overhead compared with the improved *topk* when the values of lower-k and upper-K are small. However, the *heapk* outperforms the improve *topk* for large values of lower-k and upper-K.

References

1. Huang, D., Xu, Y., Trotman, A., Geva, S.: Overview of inex 2007 link the wiki track. In Fuhr, N., Kamps, J., Lalmas, M., Trotman, A., eds.: Focused Access to

XML Documents. Volume 4862 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 373–387
2. Geva, S.: Gpx: Ad-hoc queries and automated link discovery in the wikipedia. In Fuhr, N., Kamps, J., Lalmas, M., Trotman, A., eds.: Focused Access to XML Documents. Volume 4862 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 404–416
3. Porter, M.: An algorithm for suffix stripping. Program **14**(3) (1980) 130–137
4. Spärck Jones, K.: Automatic Keyword Classification for Information Retrieval. Archon Books (1971)
5. Xu, J., Croft, W.B.: Corpus-based stemming using cooccurrence of word variants. ACM Trans. Inf. Syst. **16**(1) (1998) 61–81
6. Jia, X.F., Trotman, A., O'Keefe, R.: Efficient accumulator initialisation. In: Proceedings of the 15th Australasian Document Computing Symposium (ADCS2010), Melbourne, Australia (2010)
7. Trotman, A.: Compressing inverted files. Inf. Retr. **6**(1) (2003) 5–19
8. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. **8**(1) (2005) 151–166
9. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, ACM (2007) 183–190
10. Jia, X.f., Trotman, A., O'Keefe, R., Huang, Z.: Application-specific disk I/O optimisation for a search engine. In: PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, Washington, DC, USA, IEEE Computer Society (2008) 399–404
11. Buckley, C., Lewit, A.F.: Optimization of inverted vector searches. (1985) 97–110
12. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. **14**(4) (1996) 349–379
13. Tsegay, Y., Turpin, A., Zobel, J.: Dynamic index pruning for effective caching. (2007) 987–990
14. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. J. Am. Soc. Inf. Sci. **47**(10) (1996) 749–764
15. Anh, V.N., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination. (2001) 35–42
16. Trotman, A., Jia, X.F., Geva, S.: Fast and effective focused retrieval. In Geva, S., Kamps, J., Trotman, A., eds.: Focused Retrieval and Evaluation. Volume 6203 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 229–241
17. Bentley, J.L., Mcilroy, M.D.: Engineering a sort function (1993)
18. Persin, M.: Document filtering for fast ranking. (1994) 339–348
19. Moffat, A., Zobel, J., Sacks-Davis, R.: Memory efficient ranking. Inf. Process. Manage. **30**(6) (1994) 733–744
20. Moffat, A., Zobel, J., Klein, S.T.: Improved inverted file processing for large text databases. (1995) 162–171
21. Anh, V.N., Moffat, A.: Random access compressed inverted files. Australian Computer Science Comm.: Proc. 9th Australasian Database Conf., ADC **20**(2) (February 1998) 1–12
22. Anh, V.N., Moffat, A.: Compressed inverted files with reduced decoding overheads. (1998) 290–297
23. Schenkel, R., Suchanek, F., Kasneci, G.: YAWN: A semantically annotated wikipedia xml corpus. (March 2007)

24. Amati, G., Van Rijsbergen, C.J.: Probabilistic models of information retrieval based on measuring the divergence from randomness. ACM Trans. Inf. Syst. **20**(4) (2002) 357–389