Top-k Ranked Document Search in General Text Databases

J. Shane Culpepper[1], Gonzalo Navarro[2⋆], Simon J. Puglisi[1], and Andrew Turpin[1]

[1] School of Computer Science and Information Technology, RMIT Univ., Australia
{shane.culpepper,simon.puglisi,andrew.turpin}@rmit.edu.au
[2] Department of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. Text search engines return a set of k documents ranked by similarity to a query. Typically, documents and queries are drawn from natural language text, which can readily be partitioned into words, allowing optimizations of data structures and algorithms for ranking. However, in many new search domains (DNA, multimedia, OCR texts, Far East languages) there is often no obvious definition of words and traditional indexing approaches are not so easily adapted, or break down entirely. We present two new algorithms for ranking documents against a query without making any assumptions on the structure of the underlying text. We build on existing theoretical techniques, which we have implemented and compared empirically with new approaches introduced in this paper. Our best approach is significantly faster than existing methods in RAM, and is even three times faster than a state-of-the-art inverted file implementation for English text when word queries are issued.

1 Introduction

Text search is a vital enabling technology in the information age. Web search engines such as Google allow users to find relevant information quickly and easily in a large corpus of text, $\mathcal{T}$. Typically, a user provides a query as a list of words, and the information retrieval (IR) system returns a list of *relevant* documents from $\mathcal{T}$, ranked by *similarity*.

Most IR systems rely on the *inverted index* data structure to support efficient relevance ranking [24]. Inverted indexes require the definition of *terms* in $\mathcal{T}$ prior to their construction. In the case of many natural languages, the choice of terms is simply the vocabulary of the language: words. In turn, for the inverted index to operate efficiently, queries must be composed only of terms that are in the index. For many natural languages this is intuitive for users; they can express their information needs as bags of words or phrases.

However, in many new search domains the requirement to choose terms prior to indexing is either not easily accomodated, or leads to unacceptable restrictions on queries. For example, several Far East languages are not easily parsed into

⋆ Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

words, and a user may adopt a different parsing as that used to create the index. Likewise, natural language text derived from OCR or speech-to-text systems may contain "words" that will not form terms in the mind of a user because they contain errors. Other types of text simply do not have a standard definition of a term, such as biological sequences (DNA, protein) and multimedia signals.

With this in mind, in this paper we take the view of a text database (or collection) $\mathcal{T}$ as a string of n symbols drawn from an alphabet Σ. $\mathcal{T}$ is partitioned into N documents $\{d_1, d_2, \ldots, d_N\}$. Queries are also strings (or sets of strings) composed of symbols drawn from Σ. Here, the symbols in Σ may be bytes, letters, nucleotides, or even words if we so desire; and the documents may be articles, chromosomes or any other texts in which we need to search. In this setting we consider the following two problems.

Problem 1. A *document listing search* takes a query $q \in \Sigma^*$ and a text $\mathcal{T} \in \Sigma^*$ that is partitioned into N documents, $\{d_1, d_2, \ldots, d_N\}$, and returns a list of the documents in which q appears at least once.

Problem 2. A *ranked document search* takes a query $q \in \Sigma^*$, an integer $0 < k \leq N$, and a text $\mathcal{T} \in \Sigma^*$ that is partitioned into N documents $\{d_1, d_2, \ldots, d_N\}$, and returns the top-$k$ documents ordered by a similarity measure $\hat{S}(q, d_i)$.

By generalizing the problems away from words, we aim to develop indexes that support efficient search in new types of text collections, such as those outlined above, while simultaneously enabling users in traditional domains (like web search) to formulate richer queries, for example containing partial words or markup. In the ranked document search problem, we focus on the specific case where $\hat{S}(q, d_i)$ is the TF×IDF measure. TF×IDF is the basic building block for a large class of similarity measures used in the most successful IR systems.

This paper contains two contributions towards efficient ranked document search in general texts. (1) We implement, empirically validate and compare existing theoretical proposals for document listing search on general texts, and include a new variant of our own. (2) We propose two novel algorithms for ranked document search using general query patterns. These are evaluated and compared empirically to demonstrate that they perform more efficiently than document listing approaches. In fact, the new ranked document search algorithms are three times faster than a highly tuned inverted file implementation that assumes terms to be English words.

Our approach is to build data structures that allow us to efficiently calculate the frequency of a query pattern in a document (TF) *on the fly*, unlike traditional inverted indexes that stores precomputed TF values for specific query patterns (usually words). Importantly, we are able to derive this TF information in an order which allows rapid identification of the top-k ranked documents. We see this work as an important first step toward practical ranked retrieval for large general-text collections, and an extension of current indexing methods beyond traditional algorithms that assume a lexicon of terms *a priori*.

2 Basic Concepts

Relevance Ranking. We will focus on the TF×IDF measure, where $\text{TF}_{t,d}$ is the number of times term t appears in document d, and IDF_t is related to the number of documents where t appears.

Suffix Arrays and Self-Indexes. The suffix array $A[1..n]$ of a text collection $\mathcal{T}$ of length n is a permutation of $(1 \dots n)$, so that the suffixes of $\mathcal{T}$, starting at the consecutive positions indicated in A, are lexicographically sorted [10]: $\mathcal{T}[A[i]..n] < \mathcal{T}[A[i+1]..n]$. Because of the lexicographic ordering, all the suffixes starting with a given substring t of $\mathcal{T}$ form a range $A[sp..ep]$, which can be determined by binary search in $O(|t| \log n)$ time. Variants of this basic suffix array are efficient data structures for returning all positions in $\mathcal{T}$ where a query pattern q occurs; once sp and ep are located for $t = q$, it is simple to enumerate the $occ = ep - sp + 1$ occurrences of q. However, if $\mathcal{T}$ is partitioned into documents, then listing the documents that contain q, rather than all occurrences, in less than $O(occ)$ time is not so straightforward; see Section 3.1.

Self-indexes [13] offer the same functionality as a suffix array but are heavily compressed. More formally, they can (1) extract any text substring $\mathcal{T}[i..j]$, (2) compute sp and ep for a pattern t, and (3) return $A[i]$ for any i.

For example, the Alphabet-Friendly FM-index (AF-FMI) [5] occupies $nH_h(\mathcal{T}) + o(n \log \sigma)$ bits, where σ is the size of the text alphabet, H_h is the h-th order empirical entropy [11] (a lower bound on the space required by any order-h statistical compressor), and $h \leq \alpha \log_\sigma n$ for any constant $0 < \alpha < 1$. It carries out (1) in time $O(\log^{1+\epsilon} n + (j-i) \log \sigma)$ for any constant $\epsilon > 0$, (2) in time $O(|t| \log \sigma)$ and (3) in time $O(\log^{1+\epsilon} n)$.

Wavelet Trees. The *wavelet tree* [8] is a data structure for representing a sequence $D[1..n]$ over an alphabet Σ of size σ. It requires $nH_0(D) + o(n \log \sigma) + O(\sigma \log n)$ bits of space, which is asymptotically never larger than the $n\lceil \log \sigma \rceil$ bits needed to represent D in plain form (assuming $\sigma = o(n)$), and can be significantly smaller if D is compressible. A wavelet tree computes $D[i]$ in time $O(\log \sigma)$, as well as $\text{rank}_c(D, i)$, the number of occurrences of symbol c in $D[1..i]$, and $\text{select}_c(D, j)$, the position in D of the j-th occurrence of symbol c.

An example of a wavelet tree is shown in Fig. 1, and has a structure as follows. At the root, we divide the alphabet Σ into symbols $< c$ and $\geq c$, where c is the median of Σ. Then store bitvector $B_{\text{root}}[1..n]$ in the root node, where $B_{\text{root}}[i] = 0$ if $D[i] < c$ and 1 otherwise. Now the left child of the root will handle sequence D_{left}, formed by concatenating together all the symbols $< c$ in $D[1..n]$ (respecting the order); and the right child will handle D_{right}, which has the symbols $\geq c$. At the leaves, where all the symbols of the corresponding D_{leaf} are equal, nothing is stored. It is easy to see that there are $\lceil \log \sigma \rceil$ levels and that n bits are spent per level, for a total of at most $n\lceil \log \sigma \rceil$ bits. If, instead, the bitvectors at each level are represented in compressed form [17], the total space of each bitvector B_v becomes $nH_0(B_v) + o(n)$, which adds up to the promised $H_0(D) + o(n \log \sigma) + O(\sigma \log n)$ bits for the whole wavelet tree.

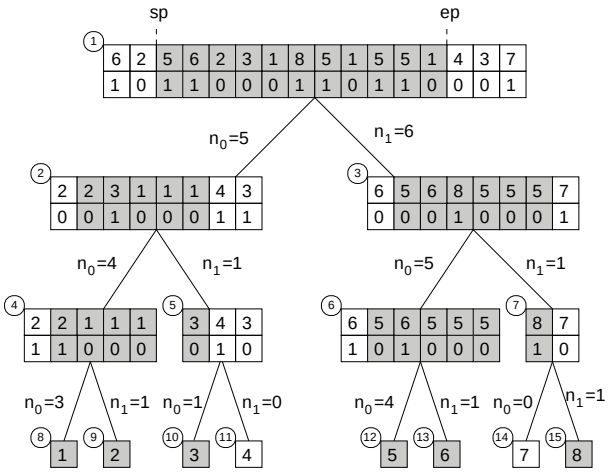

Fig. 1. $D = \{6, 2, 5, 6, 2, 3, 1, 8, 5, 1, 5, 5, 1, 4, 3, 7\}$ as a wavelet tree. The top row of each node shows D, the second row the bitvector B_v, and numbers in circles are node numbers for reference in the text. n_0 and n_1 are the number of 0 and 1 bits respectively in the shaded region of the parent node of the labelled branch. Shaded regions show the parts of the nodes that are accessed when listing documents in the region $D[sp = 3..ep = 13]$. Only the bitvectors (preprocessed for rank and select) are present in the actual structure, the numbers above each bitvector are included only to aid explanation.

The compressed bitvectors also allow us to obtain $B[i]$, and to compute rank and select, in constant time over the bitvectors, which enables the $O(\log \sigma)$-time corresponding operations on sequence D; in particular $D[i]$, $\text{rank}_c(D, i)$ and $\text{select}_c(D, j)$ all take $O(\log \sigma)$-time via simple tree traversals (see [13]).

3 Previous Work

3.1 Document Listing

The first solution to the document listing problem on general text collections [12] requires optimal $O(|q| + docc)$ time, where $docc$ is the number of documents returned; but $O(n \log n)$ bits of space, substantially more than the $n \log \sigma$ bits used by the text. It stores an array $D[1..n]$, aligned to the suffix array $A[1..n]$, so that $D[i]$ gives the document where text position $A[i]$ belongs. Another array, $C[1..n]$, stores in $C[i]$ the last occurrence of $D[i]$ in $D[1..i-1]$. Finally, a data structure is built on C to answer the range minimum query $\text{RMQ}_C(i, j) = \text{argmin}_{i \le r \le j} C[r]$ in constant time [4]. The algorithm first finds $A[sp..ep]$ in time $O(|q|)$ using the suffix tree of $\mathcal{T}$ [2]. To retrieve all of the unique values in $D[sp..ep]$, it starts with the interval $[s..e] = [sp..ep]$ and computes $i = \text{RMQ}_C(s, e)$. If $C[i] \ge sp$ it stops; otherwise it reports $D[i]$ and continues recursively with $[s..e] = [sp..i-1]$ and $[s..e] = [i+1..ep]$ (condition $C[i] \ge sp$ always refers to the original sp value). It can be shown that a different $D[i]$ value is reported at each step.

By representing D with a wavelet tree, values $C[i]$ can be calculated on demand, rather than stored explicitly [22]. This reduces the space to $|\,\mathrm{CSA}\,| + n \log N + 2n + o(n \log N)$ bits, where $|\,\mathrm{CSA}\,|$ is the size of any compressed suffix array and N is the number of documents (Section 2). The CSA is used to find $D[sp..ep]$, and then $C[i] = \mathrm{select}_{D[i]}(D, \mathrm{rank}_{D[i]}(D, i) - 1)$ is determined from the wavelet tree of D in $O(\log N)$ time. They use a compact data structure of $2n + o(n)$ bits [6] for the RMQ queries on C. If, for example, the AF-FMI is used as the compressed suffix array then the overall time to report all documents for query q is $O(|q| \log \sigma + \mathit{docc} \log N)$. With this representation, $\mathrm{TF}_{t,d} = \mathrm{rank}_d(D, ep) - \mathrm{rank}_d(D, sp - 1)$.

Gagie et al. [7] use the wavelet tree in a way that avoids RMQs on C at all. By traversing down the wavelet tree of D, while setting $sp' = \mathrm{rank}_b(B_v, sp - 1) + 1$ and $ep' = \mathrm{rank}_b(B_v, ep)$ as we descend to the left ($b = 0$) or right ($b = 1$) child of B_v, we reach each possible distinct leaf (document value) present in $D[sp, ep]$ once. To discover each successive unique d value, we first descend to the left child each time the resulting interval $[sp', ep']$ is not empty, otherwise we descend to the right child. By also trying the right child each time we have gone to the left, all the distinct successive d values in the interval are discovered. We also get $\mathrm{TF}_{t,d} = ep - sp + 1$ upon arriving at the leaf of each d. They show that it is possible to get the i-th document in the interval directly in $O(\log N)$ time. This is the approach we build upon to get our new algorithms described in Section 4.

Sadakane [20] offers a different space-time tradeoff. He builds a compressed suffix array A, and a parentheses representation of C in order to run RMQ queries on it without accessing C. Furthermore, he stores a bitvector B indicating the points of $\mathcal{T}$ where documents start. This emulates $D[i] = \mathrm{rank}_1(B, A[i])$ for document listing. The overall space is $|CSA| + 4n + o(n) + N \log \frac{n}{N}$ bits. Other $|CSA|$ bits are required in order to compute the $\mathrm{TF}_{t,d}$ values. If the AF-FMI is used as the implementation of A, the time required is $O(|q| \log \sigma + \mathit{docc} \log^{1+\epsilon} n)$.

Any document listing algorithm obtains *docc* trivially, and hence $\mathrm{IDF}_t = \log(N/\mathit{docc})$. If, however, a search algorithm is used that does not list all documents, IDF must be explicitly computed. Sadakane [20] proposes a $2n + o(n)$ bit data structure built over the suffix array to compute IDF_t for a given t.

3.2 Top-k Retrieval

In IR it is typical that only the top k ranked documents are required, for some k, as for example in Web search. There has been little theoretical work on solving this "top-k" variant of the document listing problem. Muthukrishnan [12] solves a variant where only the docc' documents that contain at least f occurrences of q ($\mathrm{TF}_{q,d} \geq f$) are reported, in time $O(|q| + \mathit{docc}')$. This requires a general data structure of $O(n \log n)$ bits, plus a specific one of $O((n/f) \log n)$ bits. This approach does not exactly solve the ranked document search problem. Recently, Hon et al. [9] extended the solution to return the top-k ranked documents in time $O(|q| + k \log k)$, while keeping $O(n \log n)$ bits of space. They also gave a compressed variant with $2|CSA| + o(n) + N \log \frac{n}{N}$ bits and $O(|q| + k \log^{4+\epsilon} n)$ query time, but its practicality is not clear.

4 New Algorithms

We introduce two new algorithms for top-k document search extending Gagie et al.'s proposal for document listing [7]. Gagie et al. introduce their method as a repeated application of the quantile$(D[sp..ep], p)$ function, which returns the p-th number in $D[sp..ep]$ if that subarray were sorted. To get the first unique document number in $D[sp..ep]$, we issue $d_1 = \text{quantile}(D[sp..ep], 1)$. To find the next value, we issue $d_2 = \text{quantile}(D[sp..ep], 1 + \text{TF}_{q,d_1})$. The j-th unique document will be $d_j = \text{quantile}\left(D[sp..ep], 1 + \sum_{i=1}^{j-1} \text{TF}_{q,d_i}\right)$, with the frequencies computed along the way as $\text{TF}_{t,d} = \text{rank}_d(D, ep) - \text{rank}_d(D, sp - 1)$. This lists the documents and their TF values in increasing document number order.

Our first contribution to improving document listing search algorithms is the observation, not made by Gagie et al., that the $\text{TF}_{q,d}$ value can be collected on the way to extracting document number d from the wavelet tree built on D. In the parent node of the leaf corresponding to d, $\text{TF}_{q,d}$ is equal to the number of 0-bits (resp. 1-bits) in $B_v[sp'..ep']$ is d's leaf is a left child (resp. right child). Thus, two wavelet tree rank operations are avoided; an important practical improvement.

We now recast Gagie et al.'s algorithm. When listing all distinct documents in $D[sp..ep]$, the algorithm of Gagie et al. can be thought of as a depth-first traversal of the wavelet tree that does not follow paths which do not lead to document numbers not occurring in $D[sp..ep]$.

Consider the example tree of Fig. 1, where we list the distinct numbers in $D[3..13]$. A depth-first traversal begins by following the leftmost path to leaf 8. As we step left to a child, we take note of the number of 0-bits in the range used in its parent node, labelled n_0 on each branch. Both n_0 and n_1 are calculated to determine if there is a document number of interest in the left and right child. As we enter leaf 8, we know that there are $n_0 = 3$ copies of document 1 in $D[3..13]$, and report this as $\text{TF}_{q,1} = 3$. Next in the depth-first traversal is leaf 9, thus we report $\text{TF}_{q,2} = 1$, the n_1 value of its parent node 5. The traversal continues, reporting $\text{TF}_{q,3} = 1$, and then moves to the right branch of the root to fetch the remainder of the documents to report.

Again, this approach produces the document numbers in increasing document number order. These can obviously be post-processed to extract the k documents with the highest $\text{TF}_{q,d}$ values by sorting the *docc* values. A more efficient approach, and our focus next, fetches the document numbers in TF order, and then only the first k are processed.

4.1 Top-k via Greedy Traversal

The approach used in this method is to prioritize the traversal of the wavelet tree nodes by the size of the range $[sp'..ep']$ in the node's bitvector. By traversing to nodes with larger ranges in a greedy fashion, we will reach the document leaves in TF order, and reach the first k leaves potentially having explored much less of the tree than we would have using a depth-first-style traversal.

We maintain a priority queue of (node, range) pairs, initialized with the single pair $(root, [sp..ep])$. The priority of a pair favors larger ranges, and ties are broken

in favor of deeper nodes. At each iteration, we remove the node $(v, [sp'..ep'])$ with largest $ep' - sp'$. If v is a leaf, then we report the corresponding document and its TF value, $ep' - sp' + 1$. Otherwise, the node is internal; if $B_v[sp'..ep']$ contains one or more 0-bits (resp. 1-bits) then at least one document to report lies on the left subtree (resp. right subtree) and so we insert the child node with an appropriate range, which will have size n_0 (resp. n_1), into the queue. Note we can insert zero to two new elements in the queue.

In the worst case, this approach will explore almost as much of the tree as would be explored during the constrained depth-first traversal of Gagie et al., and so requires $O(docc \log N)$ time. This worst case is reached when every node that is a parent of a leaf is in the queue, but only one leaf is required, e.g. when all of the documents in $D[sp..ep]$ have $\text{TF}_{q,d} = 1$.

4.2 Top-k via Quantile Probing

We now exploit the fact that in a sorted array $X[1..m]$ of document numbers, if a document d occurs more than $m/2$ times, then $X[m/2] = d$. The same argument applies for numbers with frequency $> m/4$: if they exist, they must occur at positions $m/4$, $2m/4$ or $3m/4$ in X. In general we have the following:

Observation 1 *On a sorted array $X[1..m]$, if there exists a $d \in X$ with frequency larger than $m/2^i$ then there exists at least one j such that $X[jm/2^i] = d$.*

Of course we cannot afford to fully sort $D[sp..ep]$. However, we can access the elements of $D[sp..ep]$ *as if they were sorted* using the aforementioned quantile queries [7] over the wavelet tree of D. That is, we can determine the document d with a given rank r in $D[sp..ep]$ using quantile$(D[sp..ep], r)$ in $O(\log N)$ time.

In the remainder of this section we refer to $D[sp..ep]$ as $X[1..m]$ with m a power of 2, and assume we can probe X as if it were sorted (with each probe requiring $O(\log N)$ time).

To derive a top-k listing algorithm, we apply Obs. 1 in rounds. As the algorithm proceeds, we will accumulate candidates for the top-k documents in a min-heap of at most k pairs of the form $(d, \text{TF}_{q,d})$, keyed on $\text{TF}_{q,d}$. In round 1, we determine the document d with rank $m/2$ and its frequency $\text{TF}_{q,d}$. If d does not already have an entry in the heap,[3] then we add the pair $(d, \text{TF}_{q,d})$ to the heap, with priority $\text{TF}_{q,d}$. This ends the first round. Note that the item we inserted in fact may have $\text{TF}_{q,d} \leq m/2$, but at the end of the round if a document d has $\text{TF}_{q,d} > m/2$, then it is in the heap. We continue, in round 2, to probe the elements $X[m/4]$ and $X[3m/4]$, and their frequencies $f_{X[m/4]}$ and $f_{X[3m/4]}$. If the heap contains less than k items, and does not contain an entry for $X[m/4]$, we insert $(X[m/4], f_{X[m/4]})$. Else we check the frequency of the the minimum item. If it is less than $f_{X[m/4]}$, we extract the minimum and insert $(X[m/4], f_{X[m/4]})$. We then perform the same check and update with $(X[3m/4], f_{X[3m/4]})$.

In round 2 we need not ask about the element with rank $2m/4 = m/2$, as we already probed it in round 1. To avoid reinspecting ranks, during the ith round,

[3] We can determine this easily in $O(1)$ time by maintaining a bitvector of size N.

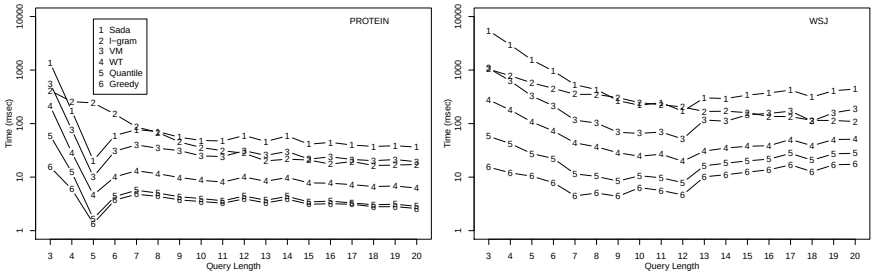

Fig. 2. Mean time to find documents for all 200 queries of each length for methods Sada, ℓ-gram, VM and WT, and mean time to report the top $k = 10$ documents by $\text{TF}_{q,d}$ for methods Quantile and Greedy. (Lines interpolated for clarity.)

we determine the elements with ranks $m/2^i, 3m/2^i, 5m/2^i$.... The total number of elements probed (and hence quantile queries) to list all documents is at most $4m/f_{min}$, where f_{min} is the k-th highest frequency in the result.

Due to Obs. 1, and because we maintain items in a min-heap, at the end of the ith round, the k most frequent documents having $\text{TF} > m/2^i$ are guaranteed to be in our heap. Thus, if the heap contains k items at the start of round $i+1$, and the smallest element in it has $\text{TF} \geq m/2^{i+1}$, then no element in the heap can be displaced; we have found the top-k items and can stop.

5 Experiments

We evaluated our new algorithms (Greedy from Section 4.1 and Quantile from Section 4.2) with English text and protein collections. We also implemented our improved version of Gagie et al.'s Wavelet Tree document listing method, labelled WT. We include three baseline methods derived from previous work on the document listing problem. The first two are implementations of Välimäki and Mäkinen [22] and Sadakane [20] as described in Section 3, labelled VM and Sada respectively. The third, ℓ-gram, is a close variant of Puglisi et al.'s inverted index of ℓ-grams [16], used with parameters $\ell = 3$ and *block size*= 4096. Our machine is a 3.0 GHz Intel Xeon with 3 GB RAM and 1024 kB on-die cache.

Experimental Data. We use two data sets. WSJ is a 100 MB collection of 36,603 news documents in text format drawn from disk three of the TREC data collection (`http://trec.nist.gov`). PROTEIN is a concatenation of 143,244 Human and Mouse protein sequences totalling 60 MB (`http://www.ebi.ac.uk/swissprot`). For each collection, a total of 200 queries of character lengths ranging from 3 to 20 which appear at least 5 times in the collection were randomly generated, for a total of 3,600 sample queries. Each query was run 10 times.

Timing Results. Fig. 2 shows the total time for 200 queries of each query length for all methods. The document listing method of Gagie et al. with our optimizations (number 4 on the graphs) is clearly the fastest method for finding

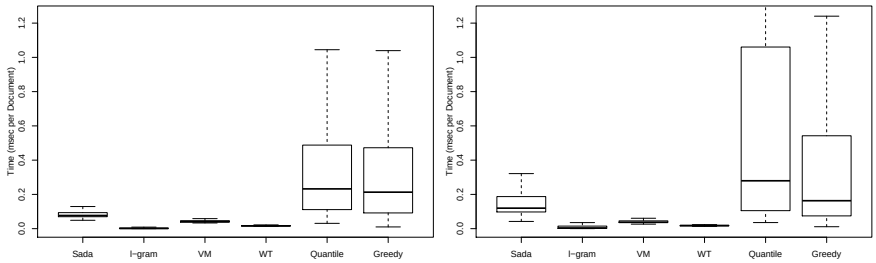

Fig. 3. Time per document listed as in Fig. 2, with 25th and 75th percentiles (boxes), median (solid line), and outliers (whiskers).

	Sada	ℓ-gram	VM	WT	Quantile	Greedy
WSJ	572	122	391	341	341	341
PROTEIN	870	77	247	217	217	217

Table 1. Peak memory use during search (MB) for the algorithms on WSJ and PROTEIN.

all documents and $\text{TF}_{q,d}$ values that contain the query q in document number order. The two algorithms which implicitly return documents in decreasing $\text{TF}_{q,d}$ order, Quantile and Greedy, are faster than all other methods. Note these final two methods are only timed to return $k = 10$ documents, but if the other methods were set the same task, their times would only increase as they must enumerate all documents prior to choosing the top-k. Moreover, we found that choosing any value of k from 1 to 100 had little effect on the runtime of Greedy and Quantile.

Note the anomalous drop in query time for $|q| = 5$ on PROTEIN for all methods except ℓ-gram. This is a result of the low *occ* and *docc* for that set of queries, thus requiring less work from the self-index methods. Time taken to identify the range $D[sp..ep]$ is very fast, and about the same for all the query lengths tested. A low *occ* value means this range is smaller, and so less work for the document listing methods. Method ℓ-gram however does not benefit from the small number of occurrences of q as it has to intersect all inverted lists for the 3-grams that make up q, which may be long even if the resulting list is short.

Fig. 3 summarizes the time per document listed, and clearly shows that the top-k methods (Quantile and Greedy) do more work per document listed. However, Fig. 2 demonstrates that this is more than recouped whenever k is small, relative to the total number of documents containing q. The average *docc* is well above 10 for all pattern lengths in the current experimental setup.

Memory Use. Table 1 shows the memory use of the methods on the two data sets. The inverted file approach, ℓ-gram, uses much less memory than the other approaches, but must have the original text available in order to filter out false matches and perform the final TF calculations. It is possible for the wavelet trees in all of the other methods to be compressed, but it is also possible to compress the text that is used (and counted) in the space requirements for

method ℓ-gram. The Sada method exhibits a higher than expected memory usage because the PROTEIN collection has a high proportion of short documents. The Sada method requires a CSA to be constructed for each document, and in this case is undesirable, as the CSA algorithm has a high startup overhead that is only recouped as the size of the text indexed increases.

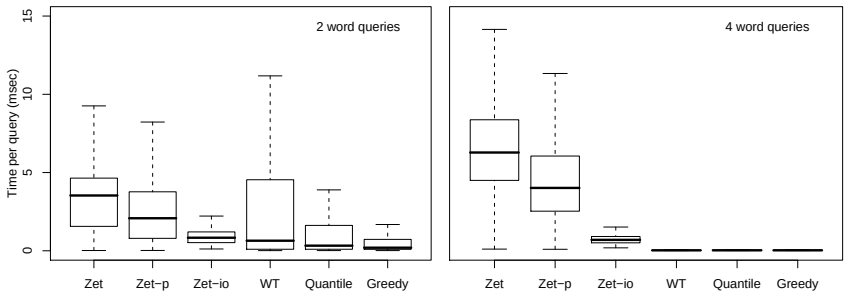

Fig. 4. Time to find word based queries using Zettair and the best of the new methods for 2 and 4 word queries on WSJ.

Term-based Search. The results up to now demonstrate that the new compressed self-index based methods are capable of performing document listing search on general patterns in memory faster than previous ℓ-gram based inverted file approaches. However, these approaches are not directly comparable to common word-based queries at which traditional inverted indexes excel. Therefore, we performed additional experiments to test if these approaches are capable of outperforming a term-based inverted file. For this sake we generated 44,693 additional queries aligned on English word boundaries from the WSJ collection.

Short of implementing an in-memory search engine, it is difficult to choose a baseline inverted file implementation that will efficiently solve the top-k document listing problem. Zettair is a publicly available, open source search engine engineered for efficiency (`www.seg.rmit.edu.au/zettair`). In addition to the usual bag-of-terms query processing using various ranking formulas, it readily supports phrase queries where the terms in q must occur in order, and also implements the impact ordering scheme of Anh and Moffat [1]. As such, we employed Zettair in three modes. Firstly, zet used the Okapi BM-25 ranking formula to return the top 20 ranked documents for the bag-of-terms q. Secondly, zet-p used the "phrase query" mode of Zettair to return the top 20 ranked documents which contained the exact phrase q. Finally, we used the zet-io mode to perform a bag-of-terms search for q using impact ordered inverted lists and the associated early termination heuristics. Zettair was modified to ensure that all efficiency measurements were done in RAM, just as the self-indexing methods require. Time to load posting lists into memory is not counted in the measurements.

Fig. 4 shows the time for searching for two- and four-word patterns. We do not show the times for Sada, VM, and ℓ-gram, as they were significantly slower than the new methods, as expected from Fig. 2. The Greedy and Quantile methods used $k = 20$. The Zet-ph has better performance, on average, than Zet, and Zet-io is the most efficient of all word-based inverted indexing methods tested. A direct comparison between the three Zettair modes and the new algorithms is tenuous, as Zettair implements a complete ranking, whereas the document listing methods simply use only the TF and IDF as their "ranking" method. However, Zet-io preorders the inverted lists to maximize efficiency, removing many of the standard calculations performed in Zet and Zet-ph. This makes Zet-io comparable with the computational cost of our new methods. The WT approach is surprisingly competitive with the best inverted indexing method, Zet-io. Given the variable efficiency of two-word queries with WT (due to the diverse number of possible document matches for each query), it is difficult to draw definitive conclusions on the relative algorithm performance. However, the Greedy algorithm is clearly more efficient than Zet-io (means 0.91ms and 0.69ms, Wilcoxon test, $p < 10^{-15}$).

When the phrase length is increased, the two standard Zettair methods get slower per query, as expected, because they now have to intersect more inverted lists to produce the final ranked result. Interestingly, all other methods get faster, as there are fewer total documents to list on average, and fewer intersections for the impact ordered inverted file. For four word queries, all of the self-indexing methods are clearly more efficient than the inverted file methods. Adding an IDF computation to Greedy and Quantile will not make them less efficient than WT.

6 Discussion

We have implemented document listing algorithms that, to date, had only been theoretical proposals. We have also improved one of the approaches, and introduced two new algorithms for the case where only the top-k documents sorted by TF values are required. For general patterns, approach WT as improved in this paper was the fastest for document listing, whereas our novel Greedy approach was much faster for fetching the top k documents (for $k < 100$, at least). In the case where the terms comprising documents and queries are fixed as words in the English language, Greedy is capable of processing 4600 queries per second, compared to the best inverted indexing method, Zet-io, which processes only 1400 on average. These results are extremely encouraging: perhaps self-index based structures can compete efficiently with inverted files. In turn, this will remove the restriction that IR system users must express their information needs as terms in the language chosen by the system, rather than in a more intuitive way.

Our methods return the top-k documents in TF order, instead of the TF×IDF order of most information retrieval systems. However, in the context of general pattern search, q only ever contains one term (the search pattern), and thus the IDF does not alter the TF order. If these data structures are to be used for bag-of-strings search, then the IDF factor may become important, and can be easily extracted using method WT, which is still faster than Zet-io in our experiments.

References

1. V. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. 29th ACM SIGIR*, pp. 372–379, 2006.
2. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.
4. M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th LATIN*, LNCS 1776, pp. 88–94, 2000.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.
6. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, pp. 459–470, 2007.
7. T. Gagie, S. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th SPIRE*, LNCS 5721, pp. 1–6, 2009.
8. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pp. 841–850, 2003.
9. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pp. 713–722, 2009.
10. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.
11. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
12. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th SODA*, pp. 657–666, 2002.
13. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
14. M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.
15. J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proc. 21th ACM SIGIR*, pp. 275–281, 1998.
16. S. Puglisi, W. Smyth, and A. Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In *Proc. 13th SPIRE*, pp. 122–133, 2006.
17. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pp. 233–242, 2002.
18. S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *JASIST*, 27:129–146, 1976.
19. S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In D. K. Harman, editor, *Proc. 3rd TREC*, 1994.
20. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
21. G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Comm. ACM*, 18(11):613–620, 1975.
22. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In B. Ma and K. Zhang, editors, *Proc. 18th CPM*, LNCS 4580, pp. 205–215, 2007.
23. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 2nd edition, 1999.
24. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):1–56, 2006.