Inf Retrieval (2008) 11:269–286
DOI 10.1007/s10791-008-9048-x

Output-sensitive autocompletion search

Holger Bast · Christian W. Mortensen · Ingmar Weber

Received: 23 January 2008 / Accepted: 23 January 2008 / Published online: 4 March 2008

Abstract We consider the following autocompletion search scenario: imagine a user of a search engine typing a query; then with every keystroke display those completions of the last query word that would lead to the best hits, and also display the best such hits. The following problem is at the core of this feature: for a fixed document collection, given a set D of documents, and an alphabetical range W of words, compute the set of all word-in-document pairs (w, d) from the collection such that $w \in W$ and $d \in D$. We present a new data structure with the help of which such autocompletion queries can be processed, on the average, in time linear in the input plus output size, independent of the size of the underlying document collection. At the same time, our data structure uses no more space than an inverted index. Actual query processing times on a large test collection correlate almost perfectly with our theoretical bound.

Keywords Autocompletion · Index data structure · Prefix search · Output-sensitive

1 Introduction

Autocompletion, in its most basic form, is the following mechanism: the user types the first few letters of some word, and either by pressing a dedicated key or automatically after each keystroke a procedure is invoked that displays all relevant words that are continuations of the typed sequence. The most prominent example of this feature is the tab-completion mechanism in a Unix shell. In the recently launched Google Suggest service completion is to frequently asked queries. Algorithmically, this basic form of autocompletion is easy: it merely requires two searches in a precompiled sorted list of strings, in order to find the first and the last completion from that list.

H. Bast · C. W. Mortensen · I. Weber (✉)
Max-Planck-Institut für Informatik, Campus E1.4, 66123 Saarbrucken, Germany
e-mail: iweber@mpi-inf.mpg.de

1.1 Problem definition

The problem we consider in this paper is derived from a more sophisticated form of autocompletion, which takes into account the *context* in which the to-be-completed word has been typed. Here, we would like an (instant) display of only those completions of the last query word which lead to hits, as well as a display of such hits. For example, if the user has typed `search autoc`, context-aware completions might be `autocomplete` and `autocompletion`, but not `autocratic`. The following definition formalizes the core problem in providing such a feature.

Definition 1 An autocompletion query is a pair (D, W), where W is a range of words (all possible completions of the last word which the user has started typing), and D is a set of documents (the hits for the preceding part of the query). To process the query means to compute the set of all word-in-document pairs (w, d) with $w \in W$ and $d \in D$.

Given an algorithm for solving autocompletion queries according to the definition above, we obtain the context-sensitive autocompletion feature as follows:

For the example query `search autoc`, W would be all words from the vocabulary starting with `autoc`, and D would be the set of all hits for the query `search`. The output would be all word-in-document pairs (w, d), where w starts with `autoc` and d contains w as well as a word starting with `search`.[1]

Now if the user continues with the last query word, e.g., `search autoco`, then we can just filter the sequence of word-in-document pairs from the previous queries, keeping only those pairs (w', d'), where w' starts with `autoc`. If, on the other hand, she starts a new query word, e.g., `search autoc pub`, then we have another autocompletion query according to Definition 1, where now W is the set of all words from the vocabulary starting with `pub`, and D is the set of all hits for `search autoc`. For the very first query word, D is the set of all documents.

In practice, we are actually interested in the *best* hits and completions for a query. This can be achieved by the following standard approach. Assume we have precomputed scores for each word-in-document pair. Given a sequence of pairs (w, d) according to Definition 1, we can then easily compute for each word w' occurring in that sequence an aggregate of the scores of all pairs (w', d) from that sequence, as well as for each document d' an aggregate of the scores of all pairs (w, d'). The precomputation of scores for word-in-document pairs such that these aggregations reflect user-perceived relevance to the given query is a much-researched area in information retrieval (Witten et al. 1999), and beyond the scope of this paper. It is for these reasons that the ranking issue is factored out of Definition 1.

To answer a series of autocompletion queries, we can obtain the new set of candidate documents D from the sequence of matching word-in-document pairs for the last query by sorting the matching (w, d) pairs. This sort takes time $O((\sum_{w \in W} |D \cap D_w|)\log(\sum_{w \in W} |D \cap D_w|))$ and would in practice be done together with the ranking of the completions and documents. The time for this sort is also included in the running times of our experiments in Sect. 6, but is dominated by the work to find all matching word-in-document pairs.

[1] We always assume an implicit prefix search, that is, we are actually interested in hits for all words *starting* with `search`, which is usually what one wants in practice. Whole-word-only matching can be enforced by introducing a special end of word symbol `$`.

A related user study (Bast and Weber 2007) shows that a new interactive and proactive user interface built on top of this autocompletion feature is (i) preferred by the users over classical Google-like search boxes and (ii) leads to more relevant documents being found.

1.2 Main result

Theorem 1 *Given a collection with n documents, m distinct words, $N \geq 2^5 \cdot m$ word-in-document pairs, and a (constant) average number of distinct words per document $L = N/n$, there is a data structure* AutoTree *with the following properties:*

(a) AutoTree *can be constructed in $O(N)$ time.*
(b) AutoTree *uses at most $N\lceil \log_2 n \rceil$ bits of space (which is the space used by an ordinary uncompressed inverted index).*[2]
(c) AutoTree *can process an autocompletion search query (D, W) (according to Definition 1) in time*

$$O((\alpha + \beta)|D| + \Phi),$$

where $\Phi = \sum_{w \in W} |D \cap D_w|$ and D_w is the set of documents containing word w. Here $\alpha = N|W|/(mn)$, which is bounded above by 1, *unless the word range is very large (e.g., when completing a single letter), and by L, regardless of assumptions about W. If we assume that the words in a document with l words are a random size-l subset of all words, β is at most* 2 *in expectation. In our experiments, β is indeed around* 2 *on the average and about* 4 *in the (rare) worst case; our analysis implies a general worst-case bound of* $\min(\log(mn/N), L_{\max})$, *where $L_{\max}$ is the maximum document length.*

Note that for constant α and β, the running time is asymptotically optimal, as it takes $\Omega(|D|)$ time to merely read in all of D and it takes $\Omega(\Phi + |W| + |D|) = \Omega(\Phi)$ time to output the result.[3] Also note that asymptotically, as the corpus grows, N, n, m and W will become large but $L_{\max}$, the maximum document length, and hence L, the average document length, can be assumed to remain bounded. In that case, alpha and beta are bounded even in the theoretical worst case. The necessary ingredients for the proof of Theorem 2 are developed in the next sections and they are finally assembled in Sect. 5.

The condition on N is a technicality and is satisfied for any realistic document collection. Details are given in Sect. 5. Intuitively speaking, the condition says that n, the number of documents grows at least as fast as m, the number of terms (assuming that L, the average document length, stays constant). This condition will guarantee that AutoTree requires less space than Basic, which can be understood intuitively as follows: Basic only needs to encode, for each word-in-document pair, a single *document* id (neglecting the small overhead for storing the list lengths and the word, a list pertains to). Thus its space requirement directly depends on the number of documents. AutoTree, as we will explain in the following sections, essentially encodes each such pair using its *word* id.

We implemented AutoTree, and in Sect. 6 show that its processing time correlates almost perfectly with the bound from Theorem 1(c) above. In that Section, we also compare it to an inverted index, its presumably closest competitor (see Sect. 1.4), which

[2] Strictly speaking, an uncompressed inverted index needs even more space, to store the list lengths.

[3] The statement about the required time to read in the (usually "random") set D tacitly assumes D is explicitly represented element-by-element. Of course, for the first prefix, when D is the set of all documents, this is not the case.

AutoTree outperforms by a factor of 10 in worst-case processing time (which is key for an interactive feature), and by a factor of 4 in average-case processing time.

1.3 Related work

This article is an extended version of (Bast et al. 2006), with full proofs and an extended set of experiments. The problem from Definition 1 is at the core of the CompleteSearch engine, which we have devised and implemented, and which is described in (Bast and Weber 2006); for a list of available demos, see http://search.mpi-inf.mpg.de.

In the same paper, an alternative data structure (called "HYB") for processing autocompletion queries is presented, and it is this data structure which underlies the CompleteSearch engine. Historically, AutoTree was developed over a year *before* HYB, even if this is not reflected in the order of the relevant publications. HYB focuses on compressibility (of the data), and locality of access (of the query algorithm), and is not an output-sensitive algorithm. A direct comparison between the two data structures is given in (Weber 2007). This comparison shows that, in a general setting, HYB is faster than AutoTree by a factor of roughly 2. Still, there are queries, in particular when $|W|$ is large and $|D|$ is small, for which AutoTree dominates HYB by almost an order of magnitude. Such queries arise naturally in certain applications, for example, the faceted search described in (Weber 2007). The design of a data structure which simultaneously achieves the best of both AutoTree (output-sensitivity) and HYB (locality of access and compressibility) remains a core open problem of this line of research.

The most straightforward way to process an autocompletion query (D, W) would be to explicitly search each document from D for occurrences of a word from W. However, this would give us a non-constant query processing time per element of D, completely independent of the respective $|W|$ or output size $\Phi = \sum_{w \in W} |D \cap D_w|$. For these reasons, we do not consider this approach further in this paper. Instead, our baseline in this paper is based on an inverted index, the data structure underlying most (if not all) large-scale commercial search engines (Witten et al. 1999); see Sect. 1.4.

Definition 1 looks reminiscent of multi-dimensional search problems, where the collections consists of tuples (of some fixed dimensionality), and queries are asking for all tuples contained in a tuple of given ranges (Gaede and Günther 1998; Arge et al. 1999; Ferragina et al. 2003; Alstrup et al. 2000). Provided that we are willing to limit the number of query words, such data structures could indeed be used to process our autocompletion queries. If we want fast processing times, however, any of the known data structures uses space on the order of N^{1+d}, where N is the number of word-in-document pairs in the collection, and d grows (fast) with the dimensionality. In the description of our data structures we will point out some interesting analogies to the geometric range-search data structures from (Chazelle 1988) and (McCreight 1985).

When searching for prefixes (or arbitrary patterns) in a text collection, suffix arrays are a standard choice (Manber and Myers 1990; Grossi and Vitter 2000; Grossi et al. 2004). Although these approaches are not directly applicable to our autocompletion problem, we could indeed use suffix arrays to produce the list of all documents that contain words with a given prefix (or even infix). This list could then be intersected with the set D. This, however, does not give output-sensitive behavior. In fact, this is similar to our Basic scheme, described in the next section, which computes the list of documents matching a given prefix by merging a number of precomputed lists, one for each word starting with the prefix.

The reason we have taken BASIC as our baseline, and not an algorithm based on suffix arrays, as just outlined, is as follows. Uncompressed suffix arrays use too much space, as they index every character of the collection.[4] Compressed suffix arrays are not competitive with respect to running time when it comes to *reporting* and not just *counting* the occurrences of an infix, because each reported occurrence requires a non-constant number of operations and typically incurs at least one cache miss. Experimental evidence for this is given in (Puglisi et al. 2006). We also ran a direct head-to-head comparison between suffix arrays (SSA2, Mäkinen and Navarro 2004) and our AUTOTREE data structure, where both index structures were used to give the list of documents containing a certain prefix. It again confirmed that, for the same space consumption[5] and even when $|D| = n$, suffix arrays are not superior to AUTOTREE in terms of the time requirements for this task. Furthermore, suffix arrays do not allow an easy integration of scores, which is crucial for our search engine setting. Details on the experimental comparison are given in (Weber 2007).

Note that the situation would be different if we wanted context-sensitive infix search. Suffix arrays would give that just as easily as prefix search, but neither AUTOTREE nor the inverted index easily adapt to that scenario without a significant blowup in either space consumption or query time. However, the application behind our problem definition really calls for prefix search and *not* for infix search. Infix search would return too many, mostly irrelevant matches. For example, when typing "search aut", we are most certainly not looking for completions like "flautist" or "aeronautics". (On the other hand, our algorithm can be easily extended to consider reasonable subwords like the "vector" in "eigenvector"; we can simply add these to the index without increasing the total index size considerably.)

Our data structure is reminiscent of wavelet trees (Grossi et al. 2003; Ferragina et al. 2006). A wavelet tree consists of a tree, built over a fixed alphabet, where each node contains a bitvector. These bitvectors are "relative" as the bits in the left/right child of a bit vector in a node correspond to the 1/0 bits in its parent. So the length of a particular bit vector depends on the number of 1/0 bits of its parent node. To allow for constant-time rank and select operations on these bit vectors, auxiliary data structures are built (Munro 1996). Our data structure also makes use of relative bitvectors, but these serve a different purpose than in wavelet trees: in our tree both children of a node store only information corresponding to the 1 bits of their parent node, and *nothing* for 0 bits. Furthermore, an integral part of our data structure is a "witness" stored by each 1 bit (whereas in a wavelet tree one only obtains the final information after descending to the leaf level).

There is a large body of more applied work on algorithms and mechanisms for *predicting* user input, for example, for typing messages with a mobile phone, for users with disabilities concerning typing, or for the composition of standard letters (Jakobsson 1986; Darragh et al. 1990; Stocky et al. 2004; Bickel et al. 2005). In (Finkelstein et al. 2001), contextual information has been used to select promising extensions for a query; the emphasis of that paper is on the quality of the extensions, while our emphasis here is on efficiency. An interesting, somewhat related phrase-browsing feature has been presented in (Paynter et al. 2000; Nevill Manning et al. 1999); in that work, emphasis was on the identification of frequent phrases in a collection.

[4] If the number of characters in the collection is N', an uncompressed suffix array needs at least $N'\lceil\log_2(N')\rceil$ bits, which exceeds the $N\lceil\log_2(n)\rceil$ bits required for an inverted index built over the words by a factor of at least the average word length.

[5] The full text as one long string was counted toward the space requirement of AUTOTREE.

1.4 The Basic scheme and outline of the rest of the paper

The following Basic scheme is our baseline in this paper. It is based on the *inverted index* (Witten et al. 1999), for which we simply precompute for each word from the collection the list of documents containing that word. For an efficient query processing, these lists are typically sorted, and we assume a sorting by document number.

Having precomputed these lists, Basic processes an autocompletion query (D, W) very simply as follows: For each word $w \in W$, fetch the list D_w of documents that contain w, compute the intersection $D \cap D_w$, and append it to the output.

Lemma 1 Basic *uses time at least* $\Omega(\sum_{w \in W} \min\{|D|, |D_w|\})$ *to process an autocompletion query (D, W). The inverted lists can be stored using a total of at most* $N \cdot \lceil \log_2 n \rceil$ *bits, where n is the total number of documents, and N is the total number of word-in-document pairs in the collection.*

Proof Basic computes one intersection for each $w \in W$ and any algorithm for intersecting D and D_w has to differentiate between $2^{\min\{|D|,|D_w|\}}$ possible outputs. For the space usage, it suffices to observe that the elements of the inverted lists, are just a rearrangement of the sets of distinct words from all documents, and that each document number can be encoded with $\lceil \log_2 n \rceil$ bits (we do not consider issues of compression in this paper). □

Lemma 1 points out the inherent problem of Basic: its query processing time depends on the size of both $|D|$ *and* $|W|$, and it can become $|D| \cdot |W|$ in the worst case.

In the following sections, we develop a new indexing scheme AutoTree, with the properties given in Theorem 1. A combination of four main ideas will lead us to this new scheme: a tree over the words (Sect. 2), relative bit vectors (Sect. 3), pushing up the words (Sect. 4), and dividing into blocks (Sect. 5). In Sect. 6, we will complement our theoretical findings with experiments on a large test collection.

2 Building a tree over the words (TREE)

The idea behind our first scheme on the way to Theorem 1 is to *increase the amount of preprocessing by precomputing inverted lists not only for words but also for their prefixes*. More precisely, we construct a complete binary tree with m leaves, where m is the number of distinct words in the collection. We assume here and throughout the paper that m is a power of two. For each node v of the tree, we then precompute the sort ed list D_v of documents which contain at least one word from the subtree of that node. The lists of the leaves are then exactly the lists of an ordinary inverted index, and the list of an inner node is exactly the union of the lists of its two children. The list of the root node is exactly the set of all non-empty documents. A simple example is given in Fig. 1.

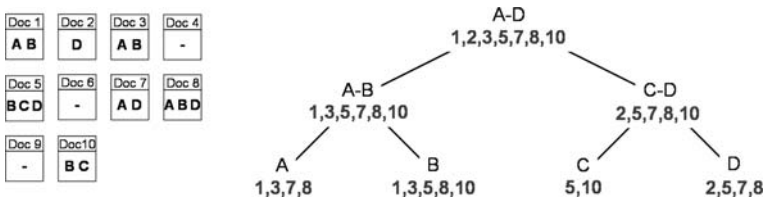

Fig. 1 Toy example for the data structure of scheme TREE with 10 documents and 4 different words

Given this tree data structure, an autocompletion query given by a word range W and a set of documents D is then processed as follows.

1. Compute the unique minimal sequence $v_1, \ldots, v_\ell$ of nodes with the property that their subtrees cover exactly the range of words W. Process these ℓ nodes from left to right, and for each node v invoke the following procedure.
2. Fetch the list D_v of v and compute the intersection $D \cap D_v$. If the intersection is empty, do nothing. If the intersection is non-empty, then if v is a leaf corresponding to word w, report for each $d \in D \cap D_v$ the pair (w, d). If v is not a leaf, invoke this procedure (step 2) recursively for each of the two children of v.

Scheme TREE can potentially save us time: If the intersection computed at an inner node v in step 2 is empty, we know that none of the words in the whole subtree of v is a completion leading to a hit, that is, *with a single intersection we are able to rule out a large number of potential completions*. However, if the intersection at v is non-empty, we know nothing more than that there is *at least one word* in the subtree which will lead to a hit, and we will have to examine both children recursively. The following lemma shows the potential of TREE to make the query processing time depend on the output size instead of on W as for BASIC. Since TREE is just a step on the way to our final scheme AUTOTREE, we do not give the exact query processing time here but just the number of nodes visited, because we need exactly this information in the next section (Fig. 2).

Lemma 2 *When processing an autocompletion query (D, W) with TREE, at most* $2(|W'| + 1)\log_2|W|$ *nodes are visited, where* W' *is the set of all words from W that occur in at least one document from D.*

Proof A node at height h has at most 2^h nodes below it. So each of the nodes $v_1,\ldots,v_l$ has height at most $\lfloor \log_2|W| \rfloor$. Further, no three nodes from $v_1,\ldots,v_l$ have identical height, which implies that $l \leq 2\lfloor \log|W| \rfloor$. Similarly, for each word in W' we need to visit at most two additional nodes, each at height below $\lfloor \log|W| \rfloor$. □

The price TREE pays in terms of space is large. In the worst case, each level of the tree would use just as much space as the inverted index stored at the leaf level, which would give a blow-up factor of $\log_2 m$.

3 Relative bitvectors (TREE+BITVEC)

In this section, we describe and analyze TREE+BITVEC, which reduces the space usage from the last section, while maintaining as much as possible of its potential for a query processing time depending on W', the set of matching completions, instead of on W (Sect. 2). *The basic trick will be to store the inverted lists via relative bit vectors*. The

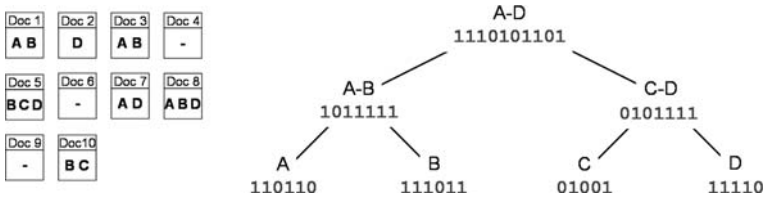

Fig. 2 The data structure of TREE+BITVEC for the toy collection from Fig. 1

resulting data structure turns out to have similarities with the static 2-dimensional orthogonal range counting structure of Chazelle (Chazelle 1988).

In the root node, the list of all non-empty documents is stored as a bit vector: when N is the number of documents, there are N consecutive bits, and the ith bit corresponds to document number i, and the bit is set to 1 if and only if that document contains at least one word from the subtree of the node. In the case of the root node this means that the ith bit is 1 if and only if document number i contains any word at all.

Now consider any one child v of the root node, and with it store a vector of N' bits, were N' is the number of 1-bits in the parent's bit vector. To make it interesting already at this point in the tree, assume that indeed some documents are empty, so that not all bits of the parent's bit vector are set to one, and $N' < N$. Now the jth bit of v corresponds to the jth 1-bit of its parent, which in turn corresponds to a document number i_j. We then set the jth bit of v to 1 if and only if document number i_j contains a word in the subtree of v.

The same principle is now used for every node v that is not the root. Constructing these bit vectors is relatively straightforward; it is part of the construction given in Sect. 4.1.

Lemma 3 *Let s_{tree} denote the total lengths of the inverted lists of algorithm TREE. The total number of bits used in the bit vectors of algorithm TREE+BITVEC is then at most $2s_{\text{tree}}$ plus the number of empty documents (which cost a 0-bit in the root each).*

Proof The lemma is a consequence of two simple observations. The first observation is that wherever there was a document number in an inverted list of algorithm TREE there is now a 1-bit in the bit vector of the same node, and this correspondence is 1−1. The total number of 1-bits is therefore s_{tree}.

The second observation is that if a node v that is not the root has a bit corresponding to some document number i, then the parent node also has a bit corresponding to that same document, and that bit of the parent is set to 1, since otherwise node v would not have a bit corresponding to that document.

It follows that the nodes, which have a bit corresponding to a particular fixed document, form a subtree that is not necessarily complete but where each inner node has degree 2, and where 0-bits can only occur at a leaf. The total number of 0-bits pertaining to a fixed document is hence at most the total number of 1-bits for that same document plus one. Since for each document we have as many 1-bits at the leaves as there are words in the documents, the same statement holds without the plus one. □

The procedure for processing a query with TREE+BITVEC is, in principle, the same as for TREE. The only difference comes from the fact that the bit vectors, except that of the root, can only be interpreted relative to their respective parents.

To deal with this, we ensure that whenever we visit a node v, we have the set $\mathcal{I}_v$ of those positions of the bit vector stored at v that correspond to documents from the given set D, as well as the $|\mathcal{I}_v|$ numbers of those documents. For the root node, this is trivial to compute. For any other node v, $\mathcal{I}_v$ can be computed from its parent u: for each $i \in \mathcal{I}_u$, check if the ith bit of u is set to 1, if so compute the number of 1-bits at positions less than or equal to i, and add this number to the set $\mathcal{I}_v$ and store by it the number of the document from D that was stored by i. With this enhancement, we can follow the same steps as before, except that we have to ensure now that whenever we visit a node that is not the root, we have visited its parent before. The lemma below shows that we have to visit an additional number of up to $2\log_2 m$ nodes because of this.

Lemma 4 *When processing an autocompletion query (D, W) with TREE+BITVEC, at most $2(|W'|+1)\log_2|W| + 2\log_2 m$ nodes are visited, with W' defined as in Lemma 2.*

Proof By Lemma 2, at most $2(|W'| + 1)\log_2 |W|$ nodes are visited in the subtrees of the nodes $v_1,\ldots,v_l$ that cover W. It therefore remains to bound the total number of nodes contained in the paths from the root to these nodes $v_1,\ldots,v_l$.

First consider the special case, where W starts with the leftmost leaf, and extends to somewhere in the middle of the tree. Then each of the $v_1,\ldots,v_l$ is a left child of one node of the path from the root to v_l. The total number of nodes contained in the l paths from the root to each of $v_1,\ldots,v_l$ is then at most $d - 1$, where d is the depth of the tree. The same argument goes through for the symmetric case when the *range ends with the rightmost leaf*.

In the general case, where W begins at some intermediate leaf and ends at some other intermediate leaf, there is a node u such that the leftmost leaf of the range is contained in the left subtree of u and the rightmost leaf of the range is contained in the right subtree of u. By the argument from the previous paragraph, the paths from u to those nodes from $v_1,\ldots,v_l$ lying in the left subtree of u then contain at most $d_u - 1$ different nodes, where d_u is the depth of the subtree rooted at u. The same bound holds for the paths from u to the other nodes from $v_1,\ldots,v_l$, lying in the right subtree of u. Adding the length of the path from the root to u, this gives a total number of at most $2d - 3$. □

4 Pushing up the words (TREE+BITVEC+PUSHUP)

The scheme TREE+BITVEC+PUSHUP presented in this section gets rid of the $\log_2|W|$ factor in the query processing time from Lemma 4. *The idea is to modify the TREE+BITVEC data structure such that for each element of a non-empty intersection, we find a new word-in-document pair (w, d) that is part of the output*. For that we store with each single 1-bit, which indicates that a particular document contains a word from a particular range, one word from that document and that range. We do this in such a way that each word is stored only in one place for each document in which it occurs. When there is only one document, this leads to a data structure that is similar to the priority search tree of McCreight, which was designed to solve the so-called 3-sided dynamic orthogonal range-reporting problem in two dimensions (McCreight 1985).

Let us start with the root node. Each 1-bit of the bit vector of the root node corresponds to a non-empty document, and we store by that 1-bit the *lexicographically smallest* word occurring in that document. Actually, we will not store the word but rather its number, where we assume that we have numbered the words from $0,\ldots,m - 1$.

More than that, for all nodes at depth i (i.e., i edges away from the root), we omit the leading i bits of its word number, because for a fixed node these are all identical and can be computed from the position of the node in the tree. However, asymptotically this saving is not required for the space bounds in Theorem 1 as dividing the words into blocks will already give a sufficient reduction of the space needed for the word numbers.

Now consider anyone child v of the root node, which has exactly one half H of all words in its subtree. The bit vector of v will still have one bit for each 1-bit of its parent node, but the definition of a 1-bit of v is slightly different now from that for TREE+BITVEC. Consider the jth bit of the bit vector of v, which corresponds to the jth set bit of the root node, which corresponds to some document number i_j. Then this document contains at least one word—otherwise the jth bit in the root node would not have been set—and the number of the lexicographically smallest word contained is stored by that jth bit. Now, if document i_j contains other words, and at least one of these *other* words is contained in H, only then the jth bit of the bit vector of v is set to 1, and we store by that 1-bit *the lexicographically*

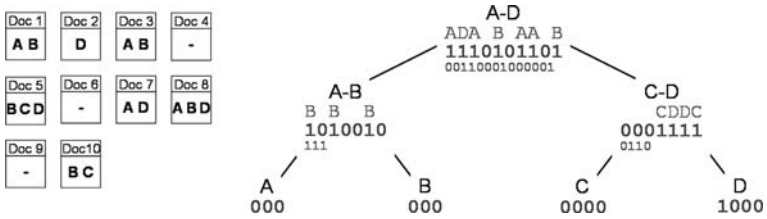

Fig. 3 The data structure of TREE+BITVEC+PUSHUP for the example collection from Fig. 1. The large bitvector in each node encodes the inverted list. The words stored by the 1-bits of that vector are shown in gray on top of the vector. The word list actually stored is shown below the vector, where A = 00, B = 01, C = 10, D = 11, and for each node the common prefix is removed, e.g., for the node marked C-D, C is encoded by 0 and D is encoded by 1. A total of 49 bits is used, not counting the redundant 000 vectors and bookkeeping information like list lengths etc.

smallest word contained in that document that has not already been stored in one of its ancestors (here only the root node).

Figure 3 explains this data structure by a simple example. The construction of the data structure is relatively straightforward and can be done in time $O(N)$. Details are given in Sect. 4.1.

To process a query we start at the root. Then, we visit nodes in such an order that whenever we visit a node v, we have the set $\mathcal{I}_v$ of exactly those positions in the bit vector of v that correspond to elements from D (and for each $i \in \mathcal{I}_v$ we know its corresponding element d_i in D). For each such position with a 1-bit, we now check whether the word w stored by that 1-bit is in W, and if so output (w, d_i). This can be implemented by random lookups into the bit vector in time $O(|\mathcal{I}_v|)$ as follows. First, it is easy to intersect D with the documents in the root node, because we can simply lookup the document numbers in the bitvector at the root. Consider then a child v of the root. What we want to do is to compute a new set I_v of document indices, which gives the numbering of the document indices of D in terms of the numbering used in v. This amounts to counting the number of 1-bits in the bitvector of v up to a given sequence of indices. Each of these so-called *rank* computations can be performed in constant time with an auxiliary data structure that uses space sublinear in the size of the bitvector (Munro 1996).

Consider again the check whether a word w stored by a 1-bit corresponding to a document from D is actually in W. This check can only fail for relatively few nodes, namely those with at least one leaf not from W in their subtree. These checks do not contribute an element to the output set, and are accounted for by the factor β mentioned in Theorem 1, and Lemmas 5 and 7 below.

Lemma 5 *With TREE+BITVEC+PUSHUP, an autocompletion query (D, W) can be processed in time* $O\big(|D| \cdot \beta + \sum_{w \in W} |D \cap D_w|\big)$, *where* β *is bounded by* $\log_2 m$ *as well as by the average number of distinct words in a document from D. For the special case, where W is the range of all words, the bound holds with* $\beta = 1$.

Proof As we noticed above, the query processing time spent in any particular node v can be made linear in the number of bits inspected via the index set $\mathcal{I}_v$. Recall that each $i \in \mathcal{I}_v$ corresponds to some document from D. Then for reasons identical to those that led to the space bound of Lemma 3, for any fixed document $d \in D$, the set of all visited nodes v which have an index in their $\mathcal{I}_v$ corresponding to d form a binary tree, and it can only happen for the leaves of that tree that the index points to a 0-bit, so that the number of these 0-bits is at most the number of 1-bits plus one.

Let again $v_1,\ldots,v_l$ denote the at most $2\log_2 m$ nodes covering the given word range W (see Sect. 2). Observe that, by the time we reach the first node from $v_1,\ldots,v_l$, the index set $\mathcal{I}_v$ will only contain indices from D', as all the 1-bits for these nodes correspond to a word in W'. Strictly speaking, this is only guaranteed after the intersection with this node, which accounts for an additional D in the total cost. Thus, each distinct word w we find in at least one of the nodes can correspond to at most $|D \cap D_w|$ 1-bits met in intersections with the bitvectors of other nodes in the set, and each 1-bit leads to at most two 0-bits met in intersections. Summing over all $w \in W$ gives the second term in the equation of the lemma.

The remaining nodes that we visit are all ancestors of one of the $v_1,\ldots,v_l$, and we have already shown in the proof of Lemma 4 that their number is at most $2\log_2 m$. Since the processing time for a node is always bounded by $O(|D|)$, that fraction of the query processing time spent in ancestors of $v_1,\ldots,v_l$ is bounded by $O(|D|\log_2 m)$. □

Lemma 6 *The bit vectors of TREE+BITVEC+PUSHUP require a total of at most* $2N + n$ *bits.*

Proof Just as for TREE+BITVEC, each 1-bit can be associated with the occurrence of a particular word in a particular document, and that correspondence is $1 - 1$. This proves that the total number of 1-bits is exactly N, and since word numbers are stored only by 1-bits and there is indeed one word number stored by each 1-bit, the total number of word numbers stored is also N. By the same argument as in Lemma 3, the number of 0-bits is at most the number of 1-bits plus 1 for each document. This can alternatively be seen as follows: Start with an empty document and, iteratively, insert the lexicographically smallest of its words, which has not been inserted yet. Each such word-in-document pair, which corresponds to a 1 in a bit vector, will be pushed up in the tree as far as possible, thereby replacing one 0-bit and creating (at most) two new 0-bits in its children. Less than two 0-bits (and in fact none at all) will only be created, if the 1-bit was already at the bottom level. □

4.1 The index construction for TREE+BITVEC+PUSHUP

The construction of the tree for algorithm TREE+BITVEC+PUSHUP is relatively straightforward and takes *constant amortized time* per word-in-document occurrence (assuming each document contains its word sorted in ascending order).

1. Process the documents in order of ascending document numbers, and for each document d do the following.
2. Process the distinct words in document d in order of ascending word number, and for each word w do the following. Maintain a *current node*, which we initialize as an artificial parent of the root node.
3. If the current node does not contain w in its subtree, then set the current node to its parent, until it does contain w in its subtree. For each node left behind in this process, append a 0-bit to the bit vector of those of its children which have not been visited. *Note: for a particular word, this operation may take non-constant time, but once we go from a node to its parent in this step, the old node will never be visited again. Since we only visit nodes, by which a word will be stored and such nodes are visited at most three times, this gives constant amortized time for this step.*
4. Set the current node to that one child which contains w in its subtree. Store the word w by this node. Add a 1-bit to the bit vector of that node.

5 Divide into blocks (TREE+BITVEC+PUSHUP+BLOCKS)

This section is our last station on the way to our main result, Theorem 1.

For a given B, with $1 \leq B \leq m$, we divide the set of all words in blocks of equal size B. We then construct the data structure according to TREE+BITVEC+ PUSHUP for each block separately. As we only have to consider those blocks, which contain any words from W, this gives a further speedup in query processing time. An autocompletion query given by a word range W and a set of documents D is then processed in the following three steps.

1. Determine the set of ℓ (consecutive) blocks, which contain at least one word from W, and for $i = 1, \ldots, \ell$, compute the subrange W_i of W that falls into block i. Note that $W = W_1 \dot{\cup} \cdots \dot{\cup} W_\ell$.
2. For $i = 1, \ldots, \ell$, process the query given by W_i and D according to TREE+ BITVEC+PUSHUP, resulting in a set of matches $M_i := \{(w, d) \in C : w \in W_i, d \in D\}$, where C is the set of word-in-document pairs.
3. Compute the union of the sets of matching word-in-document pairs $\cup_{i=1}^{\ell} M_i$ (a simple concatenation).

Lemma 7 *With TREE+BITVEC+PUSHUP+BLOCKS and block size B, an autocompletion query (D, W) can be processed in time* $O\big(|D| \cdot (\alpha + \beta) + \sum_{w \in W} |D \cap D_w|\big)$, *where* $\alpha = |W|/B$ *and* β *is bounded by* $\log_2 B$ *as well as by the average number of distinct words from* $W_1 \cup W_\ell$ *(the first and the last subrange from above) in a document from D.*

Proof Let W_i denote the subset of W pertaining to block i. Since each block contains at most B words, according to Lemma 5, we need time at most $O(|D| \log_2 B + \sum_{w \in W_i} |D \cap D_w|)$ for a block i. However, for all but at most two of these blocks (the first and the last) it holds that all words of the blocks are in W, so that according to the special case in Lemma 5, the query processing time for each of the at most $|W|/B$ inner blocks is actually $O(|D| + \sum_{w \in W_i} |D \cap D_w|)$. Summing these up gives us the bound claimed in the lemma. □

Lemma 8 *TREE+BITVEC+PUSHUP+BLOCKS with block size B requires at most* $2N + n \cdot \lceil m/B \rceil$ *bits for its bit vectors and at most* $N\lceil \log_2 B \rceil$ *bits for the word numbers stored by the* 1*-bits. For* $B \geq mn/N$, *this adds up to at most 4N for the bit vectors, and* $N(4 + \lceil \log_2 B \rceil)$ *bits in total. The auxiliary data structure (for the constant-time rank computation) requires at most an additional N bits.*

Proof To count the number of bits in the relative bitvectors, we use the same argument as for Lemma 6: there is exactly one 1-bit for each of the N word-in-document occurrences. The total number of 0-bits is at most the total number of bits in the roots of the blocks (which gives $n \cdot \lceil m/B \rceil$), plus the total number of 1-bits. So the total space for the bit vectors is bounded by $N + n\lceil m/B \rceil + N \leq 2N + n\lceil N/n \rceil \leq 4N$. The space for the word numbers is exactly $N\lceil \log_2 B \rceil$, as it requires $\lceil \log_2 B \rceil$ bits to encode a word in a block of size B. Finally, as the total length of the bit vectors is bounded by $4\,N$, we can construct the auxiliary data structure to require at most N bits (Munro 1996).[6] □

With all the required machinery in place, we can now prove Theorem 1. Part (a) of Theorem 1 is established by the construction given in Sect. 4.1. Part (b) of Theorem 1 follows from Lemma 8 by choosing $B = \lceil nm/N \rceil$. This choice of B minimizes the space

[6] Note that we do not count book keeping information (neither for AutoTree nor for Basic), such as space needed to store the lengths of the bit vectors (or the lengths of the inverted lists), as this additional space is asymptotically negligible.

bound of Lemma 8, and we call the corresponding data structure AUTOTREE. Note that it is here that we use the fact $N \geq 2^5 \cdot m$, as it ensures $5 + \log_2(nm/N) \leq \log_2 n$ and ultimately $5 + \lceil \log_2 B \rceil \leq \lceil \log_2 n \rceil$. Part (c) of Theorem 1 follows from Lemma 7 and the following remarks. If the words in a document with L words are a random size-L subset of all words, then the average number of words per document that fall into a fixed block is at most 1. In our experiments, the average value for β was 2.2.

As mentioned just before Lemma 5, β counts the number of bitvector lookup operations for a candidate document in D, which do not contribute any element to the result set. If the wordrange W spans multiple tree blocks of size $B = \lceil nm/N \rceil = \lceil m/L \rceil$, then such "useless" bitvector lookups can also occur at the root nodes of the intermediate tree blocks. However, these comparisons are accounted for by the factor α, which bounds the number of such intermediate blocks, and β only counts such bitvector lookups in the boundary blocks, which also contain at least one word not in W. Note that α is trivially bounded by the total number of blocks, which is $\lceil m/B \rceil \leq 1 + L$, which is constant.

Formally, β is defined as the number of bitvector lookups that need to be performed in the boundary blocks (of which there are at most two) for a candidate document in D until either (a) this document can be ruled out as an element of D' (as it contains no valid completions) or (b) a relevant completion is reported from this document (at which point the total number of additional bitvector lookups is bounded by twice the number of matching output elements for this document). A small, constant β thus indicates a strong output-sensitive behavior of the algorithm. Note that β is bounded by $2L_{\max}$, the maximum number of words in any document.

Finally, it remains to explain, how to obtain W' and D' from Φ. Theoretically, this can be done by having two bit vectors of lengths m and n respectively, which are to be reused for all queries. Note that the extra space required is negligible compared to the size of the data structure itself. Then, while inspecting the elements $(w, d) \in \Phi$, we set the bit corresponding to w in the bit vector of dimension m to 1, if it is not set already, and add w to the set W'. In a similar fashion, we proceed for D'. This takes time $\Theta(|\Phi|)$. Finally, to be able to reuse the two bit vectors, we pass through all elements in D' and W' and reset the corresponding bits to 0. These passes take time $O(|\Phi|)$. In the end, the elements of W' and in particular of D' will be unsorted. At first glance, this could cause a problem, as D' will be the input D for following autocompletion queries. But AUTOTREE does not require the elements of D to be sorted (unlike BASIC).

In practice, we simply sort the elements $(w, d) \in \Phi$ by w within each block, to obtain the set W'. Similarly, to obtain D', we merge the output lists of elements (w, d) for individual nodes as, if D is sorted, these will be sorted by d.[7] We chose this approach mainly as (i) it makes the use and aggregation of scores easier, (ii) we can, in fact, use the same sorting/scoring methods for documents for BASIC and AUTOTREE (which made software maintenance easier), (iii) the absolute time for sorting is small compared to the time to find the matches, and, (iv) the logarithmic factor in the time required for sorting is small compared to constant costs for, e.g., copying and other simple manipulations, so that we still obtain an almost perfect linear correlation with the size of the Φ, even as the size of Φ varies.

[7] Interestingly, the total number of such lists is bounded by $L \cdot 2^{L_{\max}}$, as each of the L blocks contributes at most $2^{L_{\max}}$ non-empty nodes. This is independent of n, m, N or W, which might seem trivial, but which is something that BASIC fails to achieve.

Table 1 The characteristics of our test collections: n = number of documents, m = number of distinct words, N/n = (rounded) average number of distinct words in a document, B^* = space-optimal choice for the block size. The last two columns give the space usage of BASIC and AUTOT(REE) in bits per word-in-document pair

Collection	Raw size	n	m	N/n	B^*	BASIC	AUTOT
ROBUST'04	1.9 GB	528,025	771,189	219	4,096	20.0	13.9
WIKIPEDIA	6.0 GB	2,363,363	7,138,267	128	65,536	22.0	17.3

6 Experiments

We tested both AUTOTREE and our baseline BASIC on two corpora. First, on the corpus of the TREC 2004 Robust Track (ROBUST'04), which consists of the documents on TREC disks 4 and 5, minus the Congressional Record (Voorhees 2004). Second, on the English (WIKIPEDIA), using only article pages and not discussion or user pages.

In both cases, we implemented AUTOTREE with an optimal block size (according to Sect. 5), which was 4096 for the ROBUST'04 collection and 65,536 for WIKIPEDIA. Block sizes were rounded to the nearest power of two.

The following Table 1 gives details on the collections and on the space consumption of the two schemes; as we can see, AUTOTREE does indeed use no more space (and for both collections, in fact, significantly less) than BASIC, as guaranteed by Theorem 1.

For the ROBUST'04 collection, queries are derived from the 200 "old"[8] queries (topics 301-450 and 601-650) of the TREC Robust Track in 2004 (Voorhees 2004). For the WIKIPEDIA collection, we generated 200 queries randomly as follows: For each query we picked a random document with uniform probability and sampled 4 terms of length at least 4 from it. Terms were sampled according to their tf-idf values, i.e., each term had a probability of being sampled proportional to $tf \cdot \log(n/df)$, where tf is the number of occurrences in the given document, n is the total number of documents and df is the number of documents containing this particular term. See Table 2 for some examples of such random queries.

In both cases, these queries were then "typed" from left to right, taking a minimum word length of 4 for the first query word, and 2 for any query word after the first. From these autocompletion queries we further omitted those, which would be obtained by simple filtering from a prefix according to the explanation following Definition 1. This filtering procedure is identical for AUTOTREE and BASIC and takes only a small fraction of the time for the autocompletion queries processed according to Definition 1, which is why we omitted it from consideration in our experiments. To give an example, for the ad hoc query `world bank criticism`, we considered the autocompletion queries `worl`, `world ba`, and `world bank cr`. For the ROBUST'04 collection, we considered a total number of 513 such autocompletion queries. For the WIKIPEDIA collection, exactly 800 such autocompletion queries were obtained (as all of the 200 "raw" queries contained exactly 4 words).

We implemented BASIC and AUTOTREE in C++ and measured query processing times on a Dual Opteron machine, with 2 Intel Xeon 3 GHz processors, 8 GB of main memory, running Linux. We measured the time for producing the output according to Definition 1. The time for scoring and ranking would be identical for AUTOTREE and BASIC, and would, according to a number of tests, take only a small fraction of the aforementioned processing time. We therefore excluded it from our measurements. For BASIC, we implemented a fast

[8] They are "old" as they had been used in previous years for TREC.

Table 2 Five of the random 200 queries generated for the WIKIPEDIA collection. From these queries, we constructed 800 autocompletion queries as described below

Query 1	Highexplosives normal pyrotechnics primarysources
Query 2	Remained growth overview Europe
Query 3	Legislatures seats typically apportion
Query 4	Salisbury inheriting westmoreland thomas
Query 5	Italy mayor frazioni baroque

Table 3 Processing times statistics of BASIC and AUTOT(REE) for all queries for both test collections. The 6th and 7th column show the kth worst processing time, where k is 10% and 5%, respectively, of the number of queries. The last column gives the correlation factor between query processing times and total list volume $\sum_{w \in W}(|D| + |D_w|)$ for BASIC, and input size plus total output volume $|D| + 10 \cdot \sum_{w \in W} |D \cap D_w|$ for AUTOTREE

Scheme	Max (s)	Mean (s)	StdDev (s)	Median (s)	90%-ile (s)	95%-ile (s)	Correl.
ROBUST'04 (513 queries)							
BASIC	14.8	0.22	0.83	0.042	0.39	0.98	0.99
AUTOT	1.15	0.07	0.12	0.042	0.17	0.24	0.99
WIKIPEDIA (800 queries)							
BASIC	71.9	2.20	7.28	0.351	4.77	10.04	0.99
AUTOT	2.17	0.17	0.25	0.032	0.47	0.63	0.99

linear-time intersect, which, in preliminary experiments not reported here, turned out to be faster than its asymptotically optimal relatives (Demaine et al. 2000).

The results from Table 3 conform nicely to our theoretical analysis. Four main observations can be made: (i) with respect to maximal query processing time, which is key for an interactive application, AUTOTREE improves over BASIC by a factor of more than 10; (ii) in average processing time, which is significant for throughput in a high-load scenario, the improvement is still a factor of 3 for the smaller collection and 13 for the larger collection; (iii) processing times of AUTOTREE are sharply concentrated around their mean, while for BASIC they vary widely (in both directions as we checked); (iv) the almost perfect correlation between query processing times and our analytical bounds (explained in the caption of Fig. 3) demonstrates both the soundness of our theoretical modeling and analysis as well as the accuracy of our implementation.

Table 4, finally, breaks down query processing times by the number of query words. As we can see, BASIC is significantly faster than AUTOTREE for the 1-word queries, however, not because AUTOTREE is slow, but because BASIC is extremely fast on these queries. This is so, because BASIC does not have to compute any intersections for a 1-word query but merely has to copy all relevant lists D_w to the output, whereas AUTOTREE has to extract, for each output element, bits from its (packed) document id and word id vectors. On multi-word queries, BASIC has to process a much larger volume than AUTOTREE, and we see essentially the situation discussed above for the overall figures.

We also experimented with AUTOTREE in the setting where the index resides *on disk*, and not in main memory as for the experiments reported above.[9] To our surprise, the average

[9] To ensure that the data was not cached by the operating system, before each experiment we read two different very large (20 GB) files from disk several times in a row. Within each experiment (running all queries for a collection), nothing was done to prevent caching by the operating system though.

Table 4 Breakdown of query processing for BASIC and AUTOTREE by number of query words

Scheme	1-word (199 queries)		Multi-word (314 queries)	
	Max (s)	Mean (s)	Max (s)	Mean (s)
ROBUST'04 (513 queries)				
BASIC	0.11	0.01	14.82	0.35
AUTOTREE	0.67	0.12	1.15	0.05
	1-word (200 queries)		Multi-word (600 queries)	
	Max (s)	Mean (s)	Max (s)	Mean (s)
WIKIPEDIA (800 queries)				
BASIC	0.23	0.03	71.85	2.92
AUTOTREE	1.36	0.41	2.17	0.09

processing time increased by only about 15%. The reason for this is that, on the average, the time for reading the whole data of a single block of AUTOTREE from disk (and each query requires data from one or at most two blocks) is dominated by the computation time of a query, most notably the time for the random bit vector lookups (see Sect. 4). That is, on the average, query processing with AUTOTREE is not IO-bound, at least not for the collection sizes we have experimented with. Since the number of operations required for a query is proportional to its input + output size, and that in turn is on average roughly proportional to the size of a block of AUTOTREE, we would expect the same behavior for larger collections too. The observations just made hold true on the average only. The smaller the output size of a query, the more IO-bound the processing is going to be. In a worst case, each step of the algorithm (inspection of a word-in-document pair) might incur one disk seek. It is an open problem how to make AUTOTREE IO-efficient also for these kinds of queries.

7 Conclusions

Motivated by a practical search application, we introduced the concept of an autocompletion query, and presented a new data structure AUTOTREE together with an algorithm to answer such queries efficiently.

We proved that our algorithm can process a given autocompletion query in time proportional to the combined input and output size, for realistic assumptions about the document collection and the query. We experimentally compared our results to those of a simple (not output-sensitive) baseline built on an inverted index. We beat this baseline by a factor of more than 10 in terms of worst-case processing time, on collections of the size of a few gigabytes. Our running times correlate almost perfectly with the predictions by our theoretical analysis.

For all our experiments, the data fit into the main memory of the machine we were using. For much larger amounts of data, locality of access would become the critical issue, and it would be meaningful to also measure the number of IO operations (Aggarwal and Vitter 1988). However, the basic operation of our algorithm is to compute the number of 1s up to a given location in a large bit vector. There is no IO-efficient data structure for this task, and hence our algorithm is not particularly IO-efficient. We remark that the same

holds for all compressed suffix array algorithms that we know of, which are all based on wavelet trees or similar ideas.

In (Bast and Weber 2006), we presented a simple scheme which achieves almost perfect locality of access, and which works well on very large data. However, this scheme is not output-sensitive—in fact, it has a considerable minimum running time independent of the output size. This is indeed a problem for a certain class of queries, and there is no easy solution. We therefore deem it a challenging and important research question to devise an algorithm for processing autocompletion queries which is both output-sensitive *and* IO-efficient.

References

Aggarwal, A., & Vitter, J. S. (1988). The input/output complexity of sorting and related problems. *Communications of the ACM, 31*(9), 1116–1127.

Alstrup, S., Brodal, G. S., & Rauhe, T. (2000). New data structures for orthogonal range searching. In *41st Symposium on Foundations of Computer Science (FOCS'00)*, pp. 198–207.

Arge, L., Samoladas, V., & Vitter, J. S. (1999). On two-dimensional indexability and optimal range search indexing. In *18th Symposium on Principles of database systems (PODS'99)*, pp. 346–357.

Bast, H., & Weber, I. (2006). Type less, find more: Fast autocompletion search with a succinct index. In *29th Conference on Research and Development in Information Retrieval (SIGIR'06)*.

Bast, H., & Weber, I. (2007). Managing helpdesk tasks with CompleteSearch: A case study. In N. Gronau (Ed.), *4th Conference on Professional Knowledge Management (WM'07)*, pp. 101–108. Potsdam, Germany. GITO.

Bast, H., Mortensen, C. W., & Weber, I. (2006). Output-sensitive autocompletion search. In *13th Symposium on String Processing and Information Retrieval (SPIRE'06)*, pp. 150–162.

Bickel, S., Haider, P., & Scheffer, T. (2005). Learning to complete sentences. In *16th European Conference on Machine Learning (ECML'05)*, pp. 497–504.

Chazelle, B. (1988). A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing, 17*(3), 427–462.

Darragh, J. J., Witten, I. H., & James, M. L. (1990). The reactive keyboard: A predictive typing aid. *IEEE Computer*, pp. 41–49.

Demaine, E. D., Lopez-Ortiz, A., & Munro, J. I. (2000). Adaptive set intersections, unions, and differences. In *11th Symposium on Discrete Algorithms (SODA'00)*, pp. 743–752.

Ferragina, P., Koudas, N., Muthukrishnan, S., & Srivastava, D. (2003). Two-dimensional substring indexing. *Journal of Computer and System Science, 66*(4), 763–774.

Ferragina, P., Giancarlo, R., & Manzini, G. (2006). The myriad virtues of wavelet trees. In *33rd International Colloquium on Automata, Languages and Programming (ICALP'06)*, pp. 560–571.

Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., & Ruppin, E. (2001). Placing search in context: The concept revisited. In *10th World Wide Web Conference (WWW'10)*, pp. 406–414.

Gaede, V., & Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys, 30*(2), 170–231.

Grossi, R., & Vitter, J. S. (2000). Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *32nd Symposium on the Theory of Computing (STOC'00)*, pp. 397–406.

Grossi, R., Gupta, A., & Vitter, J. (2003). High-order entropy-compressed text indexes. In *14th Symposium on Discrete Algorithms (SODA'03)*, pp. 841–850.

Grossi, R., Gupta, A., & Vitter, J. S. (2004). When indexing equals compression: experiments with compressing suffix arrays and applications. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 636–645, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Jakobsson, M. (1986). Autocompletion in full text transaction entry: A method for humanized input. In *Conference on Human Factors in Computing Systems (CHI'86)*, pp. 327–323.

Mäkinen, V., & Navarro, G. (2004). New search algorithms and space/time tradeoffs for succinct suffix arrays. Technical Report C-2004-20, University of Helsinki.

Manber, U., & Myers, G. (1990). Suffix arrays: A new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pp. 319–327, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

McCreight, E. M. (1985). Priority search trees. *SIAM Journal on Computing, 14*(2), 257–276.

Munro, J. I. (1996). Tables. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'96)*, pp. 37–42.

Nevill-Manning, C. G., Witten, I. H., & Paynter, G. W. (1999). Lexically-generated subject hierarchies for browsing large collections. *International Journal of Digital Libraries, 2*(2/3), 111–123.

Paynter, G. W., Witten, I. H., Cunningham, S. J., & Buchanan, G. (2000). Scalable browsing for large collections: A case study. In *5th Conference on Digital Libraries (DL'00)*, pp. 215–223.

Puglisi, S. J., Smyth, W. F., & Turpin, A. (2006). Inverted files versus suffix arrays for locating patterns in primary memory. In *SPIRE*, pp. 122–133.

Stocky, T., Faaborg, A., & Lieberman, H. (2004). A commonsense approach to predictive text entry. In *Conference on Human Factors in Computing Systems (CHI'04)*, pp. 1163–1166.

Voorhees, E. (2004). Overview of the trec 2004 robust retrieval track. In *13th Text Retrieval Conference (TREC'04)*. http://trec.nist.gov/pubs/trec13/papers/ROBUST.OVERVIEW.pdf.

Weber, I. (2007). *Efficient index structures for and applications of the CompleteSearch engine*. PhD thesis, Max Planck Institut Informatik.

Witten, I. H., Bell, T. C., & Moffat, A. (1999). *Managing gigabytes: Compressing and indexing documents and images* (2nd ed.). Morgan Kaufmann.