Efficient Top-K Fuzzy Interactive Query Expansion While Formulating a Query

From a Performance Perspective

Sigurd Wien

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Herindrasana Ramampiaro, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Interactive query expansion and fuzzy search are two efficient techniques for assisting a user in an information retrieval process. Interactive query expansion helps the user refine a query by giving suggestions on how a query might be extended to further specify the actual information need of the user. Fuzzy search, on the other hand, supports the user by including results for terms that approximately equals the query string. This avoids reformulating queries with slight misspellings and will retrieve results for indexed terms not spelled as expected. This study will look at the performance aspects of combining these concepts to give the user real time suggestions on how to complete query as the query is formulated letter by letter. These suggestions will be a set of terms from the index that are fuzzy matches of the query string terms, and are chosen based on the individual rank of the term, the semantic correlation between the individual term and the edit distance between the query and the suggestion.

The combination of these techniques is challenging from a performance aspect because each of them requires a lot of computation, and their relationship is such that these computations will be multiplicative when combined. Giving suggestions letter by letter as the user types requires a lookup for each letter and fuzzy search will expand each of these lookups with the fuzzy matches of the prefix to match against the index. For each of these different completions of the fuzzy matched prefixes, we will need to calculate the semantic correlation it has to the previous matched terms.

This study will present three algorithms to give top-k suggestions for the single term case and then extend these in three ways to handle multi term queries. These algorithms will use a trie based term index with some extensions to enable fast lookup of top-k terms that match a given prefix and to assess the semantic correlation between the terms in the suggestion. The performance review will demonstrate that our approach will be viable to use for presenting the user with suggestions in real time even with a fairly large number of terms.

Sammendrag

Interaktiv ekspansjon av spørringer og fuzzy søk er to effektive teknikker for å assistere brukere i en informasjonsgjennfinningsprosess. Interaktiv ekspansjon av spørringer hjelper brukeren å rafinere en spørring ved å gi forslag til hvordan spørringen kan bli utvidet for å mer presist spesifisere brukeres informasjonsbehov. Fuzzy søk assisterer brukeren ved å inkludere resultater for termer som er tilnærmet like spørringen brukeren har gitt til systemt. Dette gjør at man kan unngå å reformulere spørringer med små skrivefeil og gir muligheten til å motta resultater for termer i indeksen som er stavet på en annen måte enn forventet. Denne tesen vil se på ytelsesutfordringene man får når man kobinerer disse teknikkene for å i sanntid gi brukeren forslag på hvordan en spørring skal fullføres mens spørringen blir formulert bokstav for bokstav. Disse forlsagene vil bestå av et sett av termer fra indeksen som er fuzzy matcher for termene i spørringen. Disse termene er valgt på basis av deres individuelle ranking, deres inbyrdes semantiske korrelasjon og redigeringsavstanden mellom spørringen og forslaget.

Kombinasjonen av disse teknikkene er utfordrende fra et ytelsesperspektiv fordi hver og en av dem krever en stor mengde utregninger, og forholdet mellom dem gjør at disse utregningene vil oppføre seg multiplikativt når de kombineres. Det å gi et forslag til brukeren bokstav for bokstav krever et oppslag for hver bokstav og fuzzy søk vil ekspandere alle disse oppslagene med et antall fuzzy treff. For hver av disse oppslagene må det regnes ut en semantisk korrelasjon mellom termene for forslaget som skal evalueres.

Denne tesen vil presentere tre algoritmer for å finne de top-k beste forslagene for et enkeltterm tilfelle og deretter utvide disse algorithme på tre forskjellige måter for å støtte flerterms søk. Disse algoritmene vil bruke en trie basert indeks med noen utviderlser for å muliggjøre raskt oppslag av de top-k beste termene som matcher en gitt prefix og for å raskt finne semantisk korrelatsjon mellom termene i et forslag. Ytelsestestene i denne tesen vil vise at vår fremgangsmåte er en realiserbar måte å presentere brukeren med forslag i sanntid selv for gankse store indeksstørrelser.

Acknowledgements

I would like to thank my supervisor Heri Ramampiaro and my co-supervisor Øystein Torbjørnsen for their invaluable support, tips and feedback. I would also like to thank Ola Natvig for listening to some of my more crazy ideas and for giving guidance during the writing process. Last, but not least, I would like to thank my fiancé Suzanne Hansen for her unprecedented support and patience during the heaviest development of the code that are the foundation which this thesis builds on. Without her, the motivation to perfect every important corners of the code would not exist.

TABLE OF CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

With the continuing massive expansion of digital information, search engines plays a more and more important role in accessing information that satisfies a particular user's needs. The regular process of finding information consist of a user typing in a query to a search engine, which in turn returns a set of documents relevant to that query. Unfortunately, queries formulated by users are often of low quality and are one of the most important reasons for unsatisfactory search results [1]. The difficulty of formulating queries stems both from the user's inability to accurately specify their information needs, as well as their lack of knowledge of what information that resides in the search engine. This leads to nonspecific and ambiguous queries, which highly correlates with results of low precision [2].

Interactive query expansion is a technique used to alleviate this problem by suggestion continuations to a typed query, and thereby help the user formulate a better query. This may significantly increase the efficiency of the retrieval process, as Magennis and Rijsbergen (1997) [3] and Joho, Sanderson and Beaulieu (2004) [4] shows.

One does however need to be careful when implementing interactive query expansion because users are not necessarily able to choose expansion terms better than a system could in automatic query expansion as Ruthven (2003) [5] shows. The study suggest that how the user is presented with interactive query expansion is highly important for its effectiveness. Real time query expansion is a form of interactive query expansion that integrates the query expansion into the process of query formulation. White and Marchionini (2007) [6] evaluates the effectiveness of this technique. They concluded that real time query expansion was a step forward in the development of the usability of interactive query expansion by making it more frequently used, increasing the quality of initial queries and increasing user satisfaction.

Fuzzy search is another technique to support the user in the query formulation process. It enables the search engine to tolerate small misspellings in a query and retrieve results for what the query the user actually intended to write. Ji, et al. (2009) [7] outlines that this can further support users with limited knowledge of the contents of the index. For instance, names that are pronounced the same can have subtle different spellings. Words may also be slightly differently spelled in different variants of natural languages, like American and British English. Fuzzy search can in these situations recognize that the words are very similar and retrieve the correct results for the query "theater" even if the user does not know if the document of interest used the American "theater" or the British "theatre".

In this thesis, we will look at the combination of presenting the user with suggestion as the query is formulated by using interactive query expansion, similar to the real time query expansion technique, and fuzzy search. We will give the user fuzzy matched suggestions for all terms in the query strings as it is typed letter by letter. Our approach will be based entirely on the corpus in the search engine. The problem we are trying to solve is to find the k best sets of terms, given a query, from the index that can be presented as suggestions for what the user intended. A query string consists of n query terms, were the last are considered partial. A suggestion is a set of terms from the index that are fuzzy matches for the n-1 complete query terms and auto-completions for all the fuzzy matched prefixes for the partial query term. We will develop multiple techniques for deriving with these term sets efficiently.

This study will only consider the speed of the algorithms. This is a huge issue for the combination of interactive query expansion as the user types and fuzzy search. Giving the user suggestions in real time puts tight constraints on how much time we can use for each character [8]. Since we are giving suggestion letter by letter, we are essentially need to issue a query for each character the user types. Fuzzy search on the other hand tries to not only match the query string exact, but all strings that approximately matches the query string [9], which means that we may need to do a lot more than one suggestion for each character added to the query. Further, interactive query expansion require us not only to blindly auto-complete a term, but to also assess how well it correlates with the term previously entered and choose a term that are suited to continue the query. The total number of queries for a search session might therefore be very large and calculating the ranking of each of the resulting terms that make up a suggestion might be complex.

While we recognize that it is important, we will not assess the quality of the retrieved suggestions. We will however assure that the techniques we test will present the same or very similar suggestions. The work required to tune the ranking functions we present for quality is outside the scope of this thesis, but we will argue for the plausibility that this can be done by referring to similar techniques used with great effect in the literature.

This thesis consists of eight chapters. Chapter 2 will present the work that this thesis builds on. In chapter 3 we will look at the index structure that we will use to enable fast interactive fuzzy matching. Chapter 4 will present three different single term algorithms for interactive fuzzy matching. These algorithms will be extended to support multi term queries in chapter 5. Chapter 6 will describe the tests we are using to assess the performance of these algorithms, while chapter 7 will show the results of these tests and discuss their implications. We will conclude our findings and suggest further research in section 8.

2 PREVIOUS WORK

In this section we will present existing related work in the fields of query expansion and fuzzy search. They represent techniques and ideas that are important building blocks for this thesis.

2.1 QUERY EXPANSION

Cao, et al. (2008) [10] explores the use of information collected from previous queries in the current query session to increase the quality of suggestions the user receives from the search engine. They find that they are able to significantly increase the quality and coverage the suggestions with this technique.

Sadikov, et al. (2010) [11] looks into grouping query refinements into cluster that represents distinctly different information needs. This technique can be used to ensure that a variety of the possible meanings of an ambiguous query would be represented both in the result set and in the displayed query refinement suggestions. They also looks into using previous queries from the same user session similar to Cao, et al. (2008) to influence the weight of representation for the individual clusters. Their study shows that there are user perceived benefits to this technique.

Liu, Natarajan and Chen (2011) [12] explores performing query expansion by clustering the result set of a query and suggest expended queries, one for each cluster, based on the contents of each cluster. The results of the expanded queries would then be documents in the cluster, or in close proximity to the cluster. They further propose two algorithms to efficiently solve this problem.

Topic based query expansion is studied by Fan, et al. 2010 [13]. They use topic related information in their ranking algorithm to help choose appropriate suggestion on a partially completed query term given a set of previously complete query terms.

Fonseca, et al. (2005) [14] performs interactive query expansion by identifying past queries from the query logs that are closely related to a given query. It will then cluster these queries and label each cluster as a concept. These concepts are then presented to the users as way to continue the query session.

Kaczmarek (2011) [15] studies interactive query expansion with the use of clustering-by-directions. Their goal is to given the user directions in which a search can be continued after an initial query. This is done by treating the given query as a coordinate in the space of documents. It will then choose n directions from this coordinate, cluster documents around these different directions and label the directions for the user. The user can then choose some direction and retrieve the documents in that cluster.

2.2 FUZZY SEARCH

A review of different approximate string matching algorithms is given by Ukkonen (1985) [9]. Of particular interest for this study is the definition of edit operations and edit distance, and the techniques to efficiently calculate minimum edit distance.

Ukkonen (1993) [16] explores approximate string-matching over suffix trees. Multiple algorithms are devoted to do this efficiently.

Interactive fuzzy search in a trie is explored by Ji, et al. (2009) [7]. They present a single and multi-term versions of edit distance based fuzzy matching and how to efficiently retrieve the relevant documents. Their approach is based on the trie data structure and we will further build on this approach in this thesis

The techniques developed by Ji, et al. (2009) are used by Wang, et al. (2010) [17] to develop a new system to explore the Medline[1] collection with interactive fuzzy search. The result is the system iPubMed[2].

Li, Ji, et al. (2011) [18] explores type-ahead search which is a paradigm where the search engine presents results of auto-completed terms while the user types a query. They allow fuzzy matches with the trie based technique developed in Ji, et al. (2009).

Supporting efficient top-k type-ahead search is explored by Li, Wang, et al. (2012) [19]. They use priority queues to efficiently find the top-k documents related the fuzzy matches of the query. Our approach to find top-k results will be inspired by their approach.

Gram based fuzzy matching is explored by Behm, et al. (2009) [20]. While this is an efficient way of fuzzy matching to string, the incremental nature of our problem makes a trie based approach more efficient [7].

Our work differs from previous work in three essential ways. First will be looking at the combination of interactive query expansion as the user types and fuzzy search. Second we will look at building an index which has special support for some of the techniques used to rank query expansions. Third, the main focus of this study is to enable fuzzy interactive query expansion within the performance characteristics required to be truly interactive. We will therefore not attempt at enhancing the quality of the suggestions given.

[1] MEDLINE is a trademark of the U.S National Library of Medicine
(See http://www.nlm.nih.gov/databases/databases.medline.html).
[2] http://ipubmed.ics.uci.edu/

3 INDEX STRUCTURE

Fuzzy search involves not only searching for the query the user typed, but also all the possible fuzzy matches to the query terms. Combined with the frequent searches required by interactivity the share number of queries required will be quite large. To alleviate this, we propose to augment the traditional trie index with two novel features. This new data structure will enable faster access to top-k terms and an approach to efficiently calculate a measure of the semantic correlation between terms already typed and the last presumed incomplete term.

3.1 THE BASIC INDEX STRUCTURE

A trie/prefix tree is a data structure that provides efficient lookup of either a term or to find all terms with a given partial term as a prefix [21] [22]. They are typically implemented by using hash-tables, so that look-ups for a given character at a particular node is expected to be O(1), which means that a term can be matched in O(n) time, where n is the length of the term. There are also some inherent compression in this data structure, as terms with common prefixes will share the storage for that prefix.

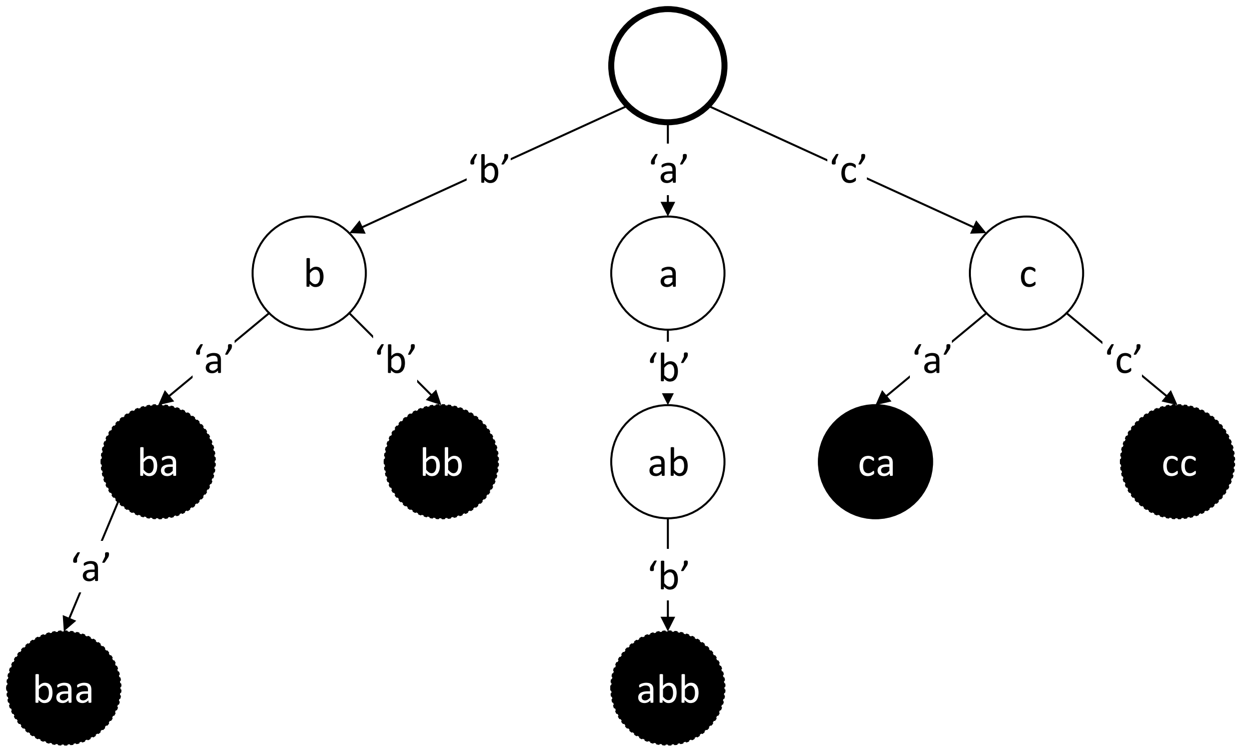


Figure 1: A plain trie with the terms "ba", "baa", "bb", "abb", "ca" and "cc" inserted.

Figure 1 show a plain trie with the terms “ba”, “baa”, “bb”, “abb”, “ca” and “cc” inserted. We use the convention here that filled nodes are terminal nodes. This variant terminates the terms at the node they are at. This allows for a node that terminates one node to also have children, like what we see for “ba” and “baa”. In many cases this arrangement might be inconvenient, and for our purpose we will use special termination nodes to represent a complete term. Every term termination node will then be a leaf node. This gives a slightly larger data structure (with a constant factor), but makes the implementations of index manipulation and querying much cleaner in our application. Each leaf node will contain an inverted index for the term the node terminates. For easier implementation of multi term queries, as will be explained below, the space character will be used as the label of the edge to the special termination nodes. Figure 2 illustrates this kind of trie with the same terms as in Figure 1.

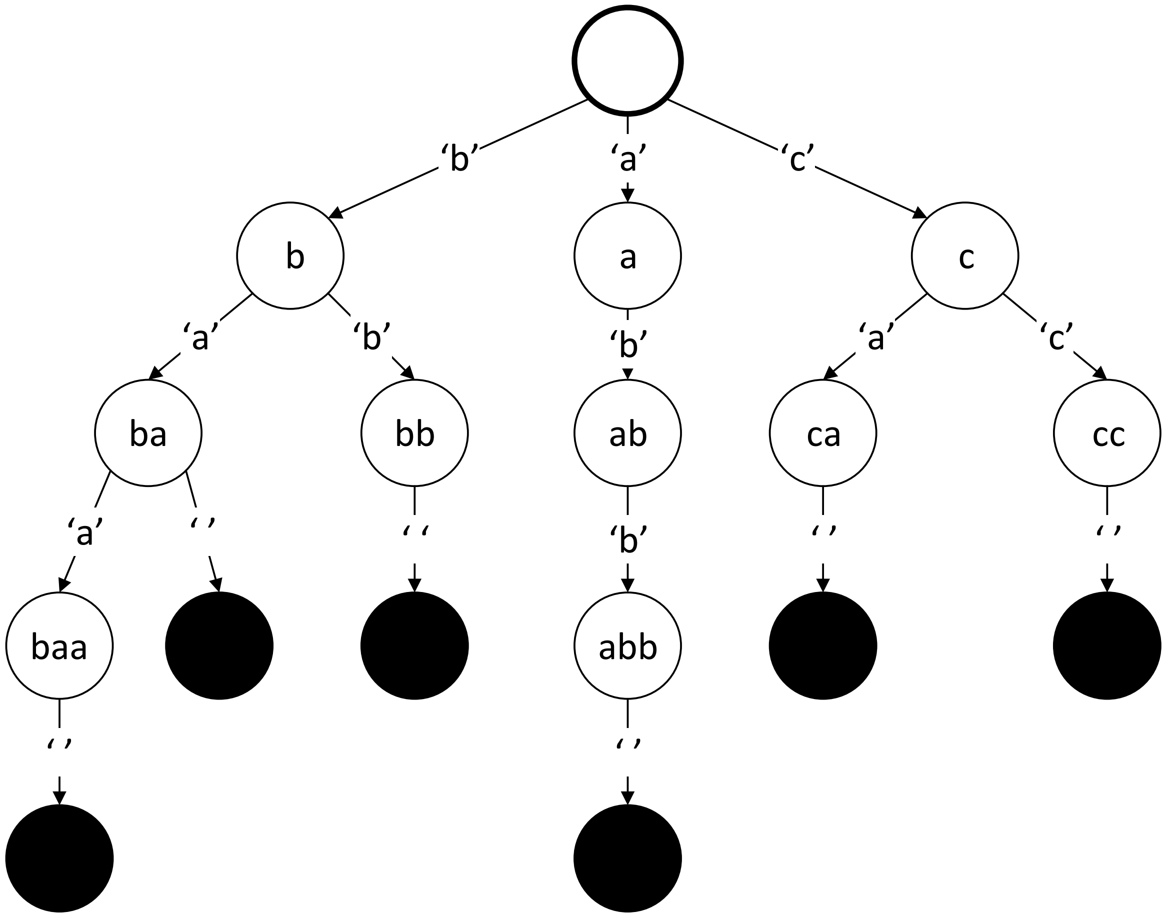


Figure 2: A trie with special termination nodes.

While this data structure are very fast for extracting suggestion for partial terms when we are only considering exact matches, our application needs to also be able to do fuzzy matches fast. To accommodate this, we augment the trie with node ranks. This rank will be derived from the terms that are inserted into the trie. The rank of a node will be the rank of the highest ranked term that it has as a descendant. This enables nodes near the root to convey some information of the value of its sub-tree, which in turn enables a much easier way to retrieve the best terms. In its implementation, a node will replace the typical hash-table of its children with a sorted list. We will refer to this data structure as a sorted trie index. Figure 3 illustrates this structure based on the index from Figure 2. The terms are here (randomly) assigned the following rank: {"baa", 0.9}. {"abb", 0.7}, {"ca", 0.6}. {"bb", 0.5}, {"cc", 0.5} and {"ba", 0.4}.

In a real application the term ranks should somehow represent the value of the terms. In our application we will use the tf-idf measure of the best document that the term is found in. The reasoning behind this gives a straight forward way to retrieve the top-n documents that corresponds to the top-k terms as the top-k terms guaranties at least to contain the top-k documents.

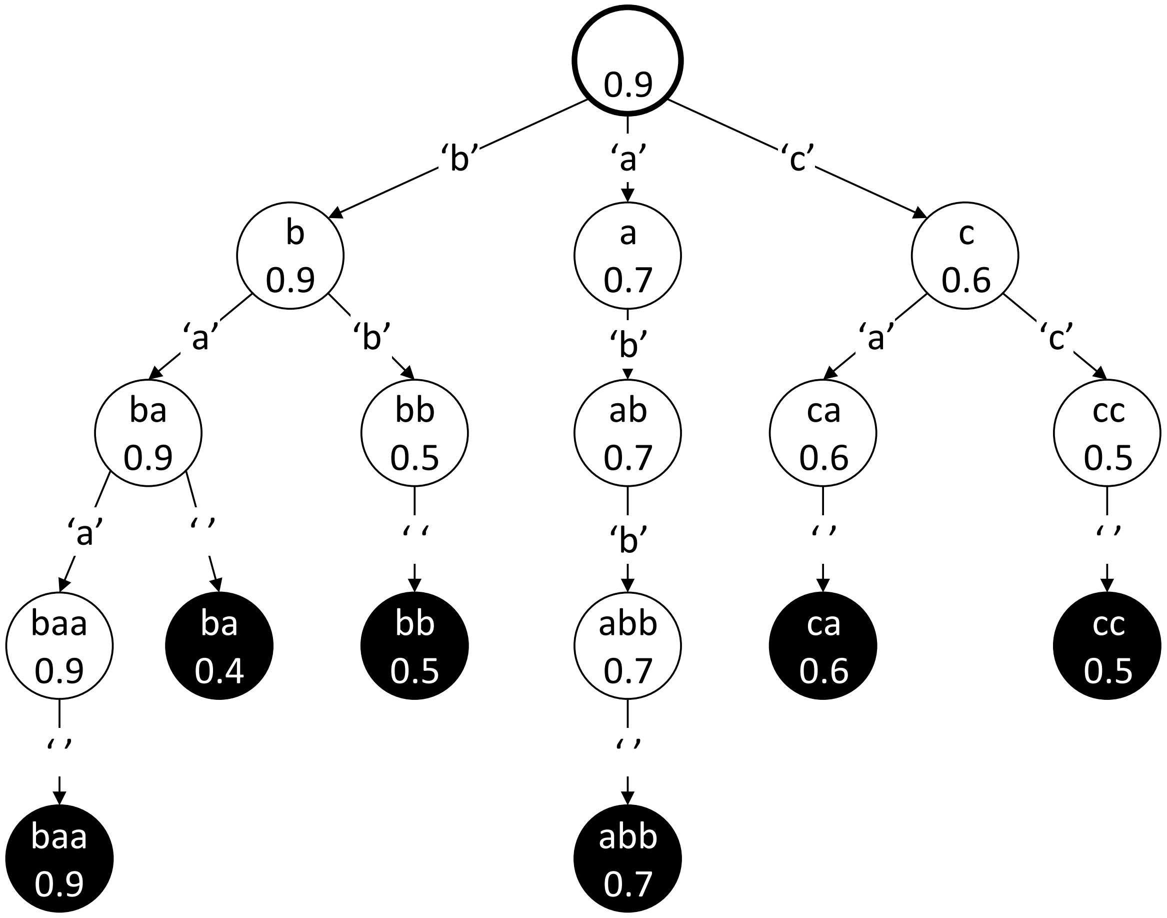


Figure 3: A trie with node ranks and children sorted by them.

3.2 Inserting terms into a sorted trie

Because every internal node needs to keep a sorted list of its children, the process of inserting, updating and deleting terms requires a bit more work than in the plain trie data structure. Figure 4 outlines the algorithm for insertions. Notice that since the list is sorted before we add, we can sort the list in O(a), where a is the size of the alphabet. The total asymptotical time usage for an insertion will then be O(an) where n is the length of the term. Updates and deletes will have the same time-usage. This algorithm will thus slow down with the size of the index, unlike for the standard trie, where the insertion is always O(n). If the set of characters are constant, which would be the case in most implementations, because of limited character sets, the time usage for the sorted trie is also O(n) for the mentioned operation (but with a larger constant factor).

```
persistent:     children = { }
                nodeCharacter

init( nodeCharacter )
        nodeCharacter = nodeCharacter

insertTerm( term, termPosting )
        insertTerm( term, termPosting, 0 )

insertTerm(term, termPosting, depth)
        TrieNode child
        if(depth == term.length)
                child = new LeafTrieNode( termPosting )
        else
                currentChar = term.charAt(depth)
                child = new InternalTrieNode( currentChar )
                child.insertTerm( term, termPosting, depth + 1 )
        children.sortedAdd( child )
        chilndre.sort()
```

Figure 4: A recursive algorithm for inserting a term into the sorted trie index.

3.3 Top-k single term suggestion retrieval

In our application, we wish to be able to retrieve terms with a certain prefix in order of their rank. In Figure 5 we have a situation where the user has typed in ‘b’ (the node is marked as grey). Here we wish to be able to retrieve {baa, 0.9}, {bb, 0.5} and {ba, 0.4} in that order. We also wish to be able to delay retrieving suggestions we do not yet know that we want. I.E we might want to retrieve all suggestions with a rank higher or equal to 0.5, and later retrieve all suggestion with a rank higher or equal that 0.4. Accomplishing this requires some bookkeeping. Figure 6 outlines and algorithm for this retrieval.

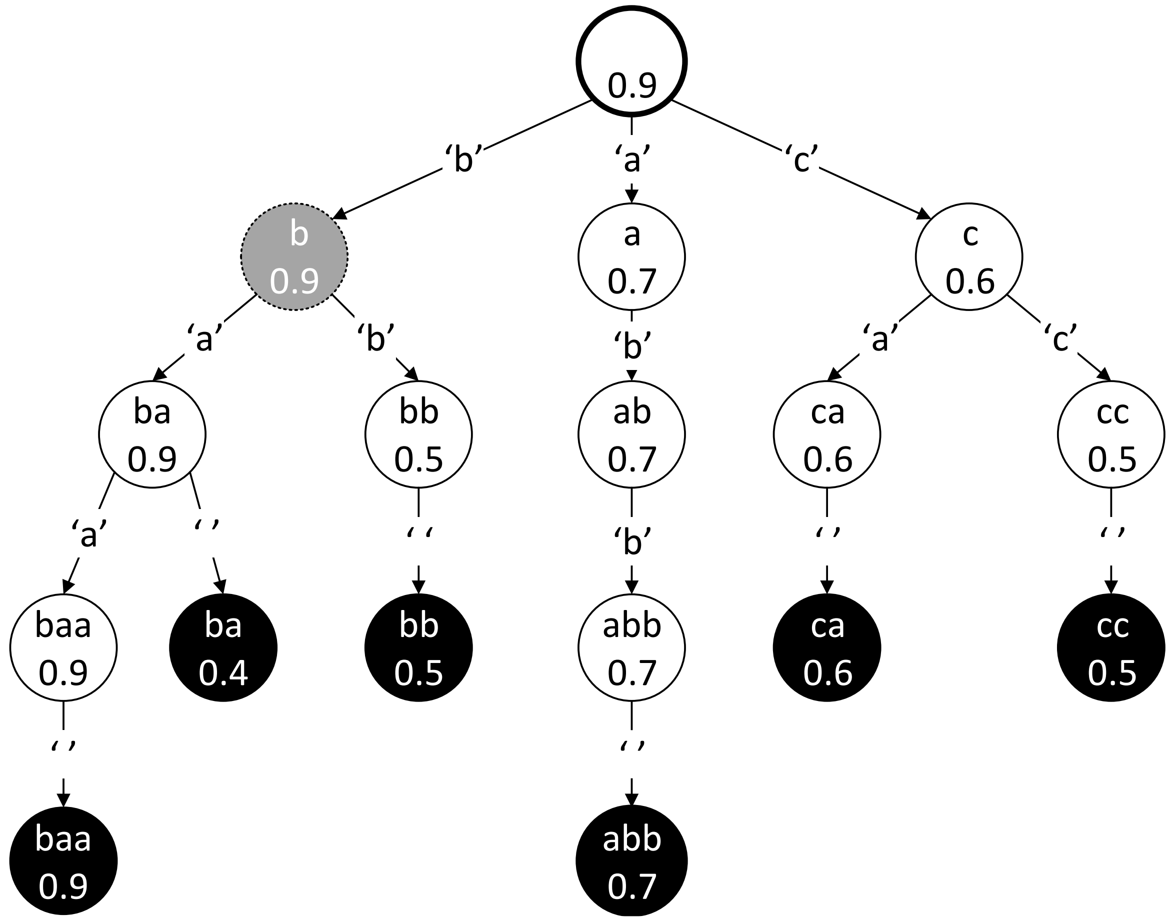


Figure 5: Querying for terms with "b" as prefix.

```
persistent:     suggestionQueue = new PriorityQueue()

initial(trieNode)
    suggestionQueue.add(trieNode)

getNextSuggestion()
        while suggestionQueue.size > 0
                currentPosition = suggestionQueue.pop()
                if currentPosition is Leaf
                        return currentPostition
                else
                        nextChild = currentPosition.getNextChild()
                        suggestionQueue.add(nextChild)
                        if currentPosition.hasMoreChildren()
                                suggestionQueue.add(currentPosition)
        return null

getNextSuggestionRank()
    if suggestionQueue.size > 0
        currentNode = suggestionQueue.peek()
        return currentNode.getRank()
    return -2
```

Figure 6: Retrieval of suggestions.

Figure 6 defines two functions. getNextSuggestion() will retrieve the next suggestion and getNextSuggestionRank() will peek the rank of the next suggestion. Internally, the currentPosition objects keeps track of which child node to explore next. This algorithm thus lets us get an arbitrary number of suggestions in order and it lets us see if we want the next suggestion or not by peeking its rank.

3.4 TERM CORRELATION CLUSTERS

While the sorted trie index will significantly speed up retrieval of top-k single term suggestions, it does not offer a complete solution in a multi-term scenario. When offering multi-terms suggestions, the correlation between the terms must also be taken into account. I.e. if the first types term was "car", then "manufacturer" would intuitively be a much more appropriate suggestion than "manuscript", even if both "manufacturer" and "manuscript" would be appropriate suggestion for the partial query "manu" and "manuscript" was ranked higher. Because we are fuzzy matching terms, we might have to consider a very large number of permutation of individual terms from the index in order to calculate which multi term suggestion that are best. However, the trie index we have developed so far makes no indication of where to find good terms with high semantic correlation. We might therefore have to do an exhaustive search to find the top-k suggestions in multi-term search, even if we can extract the top-k suggestions for each individual query term.

To solve this problem, we propose a solution inspired by the ideas of topic based interactive query expansion [13], clustering by direction based query expansion [15] and wordnets [23]. It involves finding clusters of terms that belongs to the same topic at index time, and then build multiple trie structures, one for each cluster. A term will exclusively belong to a cluster, but each term entry will contain a vector that specifies its relevancy to all clusters based on its locality among the clusters at indexing time. Figure 7 and Figure 8 illustrates one possible clustering of the terms “Camera”, “Electric”, “Car”, “Manufacturer”, “Film” and “Manuscript”.

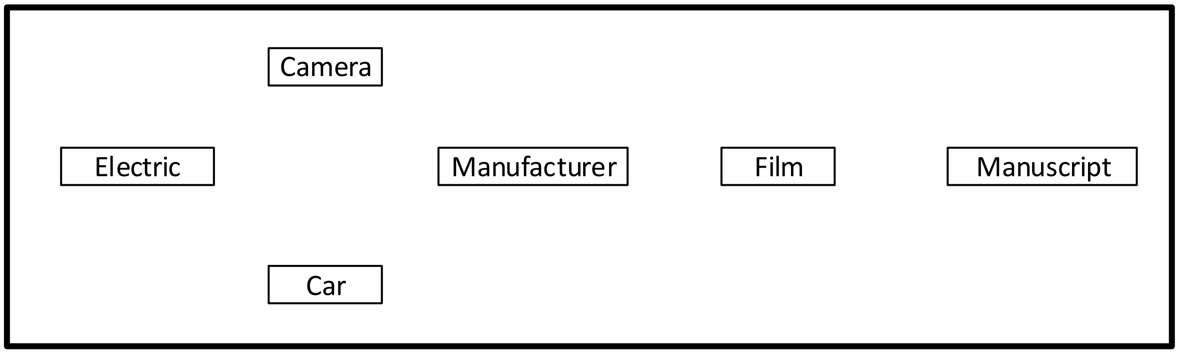


Figure 7: The terms "Camera", "Electric", "Car", "Manufacturer", "Film" and "Manuscript" in a space where more semantically correlated terms are placed closer.

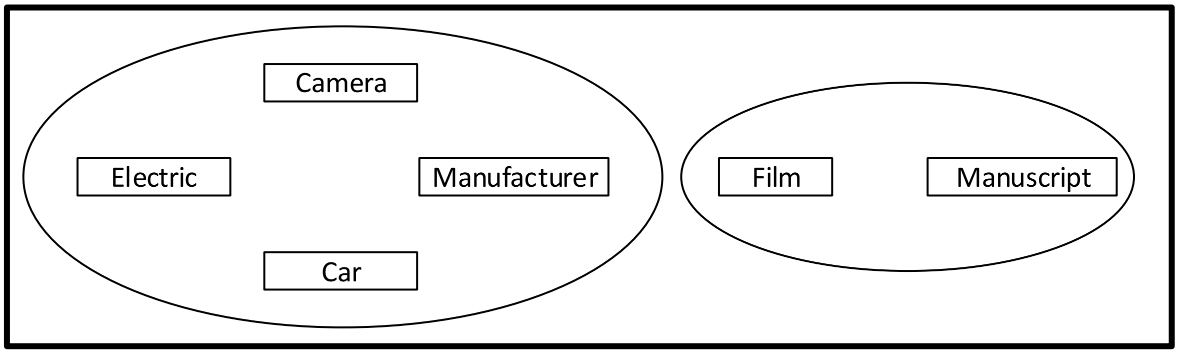


Figure 8: An example of two hard clusters on the terms from Figure 7.

When a user queries the index with the first query term, this vector of cluster relevance scores will be retrieved, and based on it, we can prioritize highly relevant clusters by just using the relevancy score as a correlation rank for all the terms in the cluster. For instance, if query the cluster in Figure 8 with the term “Manufacturer” both the displayed clusters should be emphasized about equally, since “manufacturer” is quite close to the second cluster, which is appropriate because “film manufacturer” might be a common phrase. On the other hand, “manuscript” is far away from the first cluster, so only the second cluster should be emphasized when looking for the second term.

We use the relevance for each cluster as the semantic correlation score. We will therefore avoid having to do an exhaustive search when trying to locate semantically correlated terms. The trade of here is that we lose some precision because individual terms inside the same cluster cannot be treated differently. For instance, “manufacturer manuscript” might not be a common phrase, but will be considered to be so because the second cluster is close to the term “manufacturer. This might, however, be a worthwhile tradeoff if it means that we can achieve interactive speed while still improving the quality of the results over not calculating semantic correlations at all.

Figure 9 illustrates a clustered version of the index in Figure 5, were the terms “ba” and “cc” are moved to a second cluster.

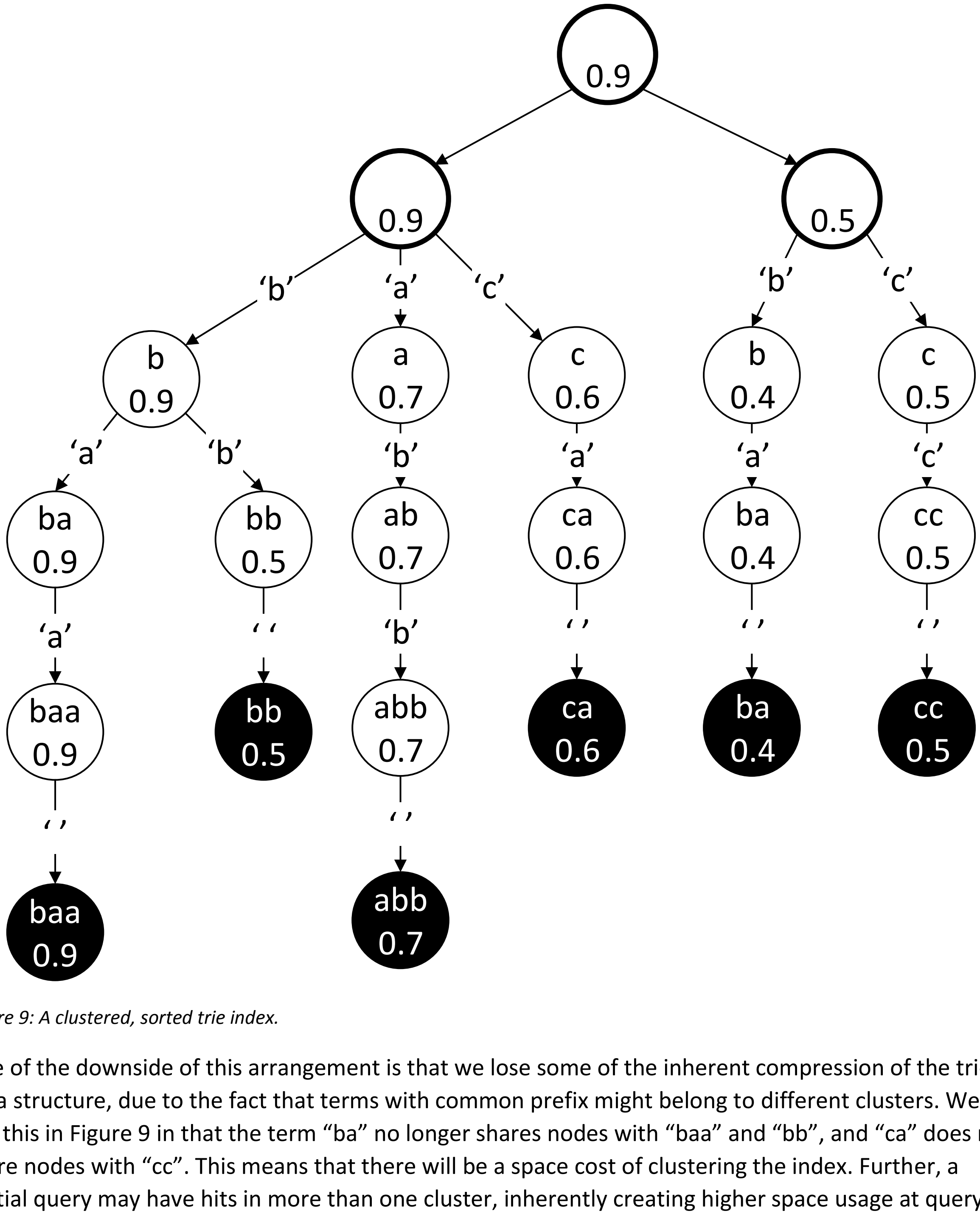


Figure 9: A clustered, sorted trie index.

One of the downside of this arrangement is that we lose some of the inherent compression of the trie data structure, due to the fact that terms with common prefix might belong to different clusters. We can see this in Figure 9 in that the term “ba” no longer shares nodes with “baa” and “bb”, and “ca” does not share nodes with “cc”. This means that there will be a space cost of clustering the index. Further, a partial query may have hits in more than one cluster, inherently creating higher space usage at query time. As an example of this, let’s revisit the example from Figure 5. The query ‘b’ now creates two active nodes, as Figure 10 illustrates.

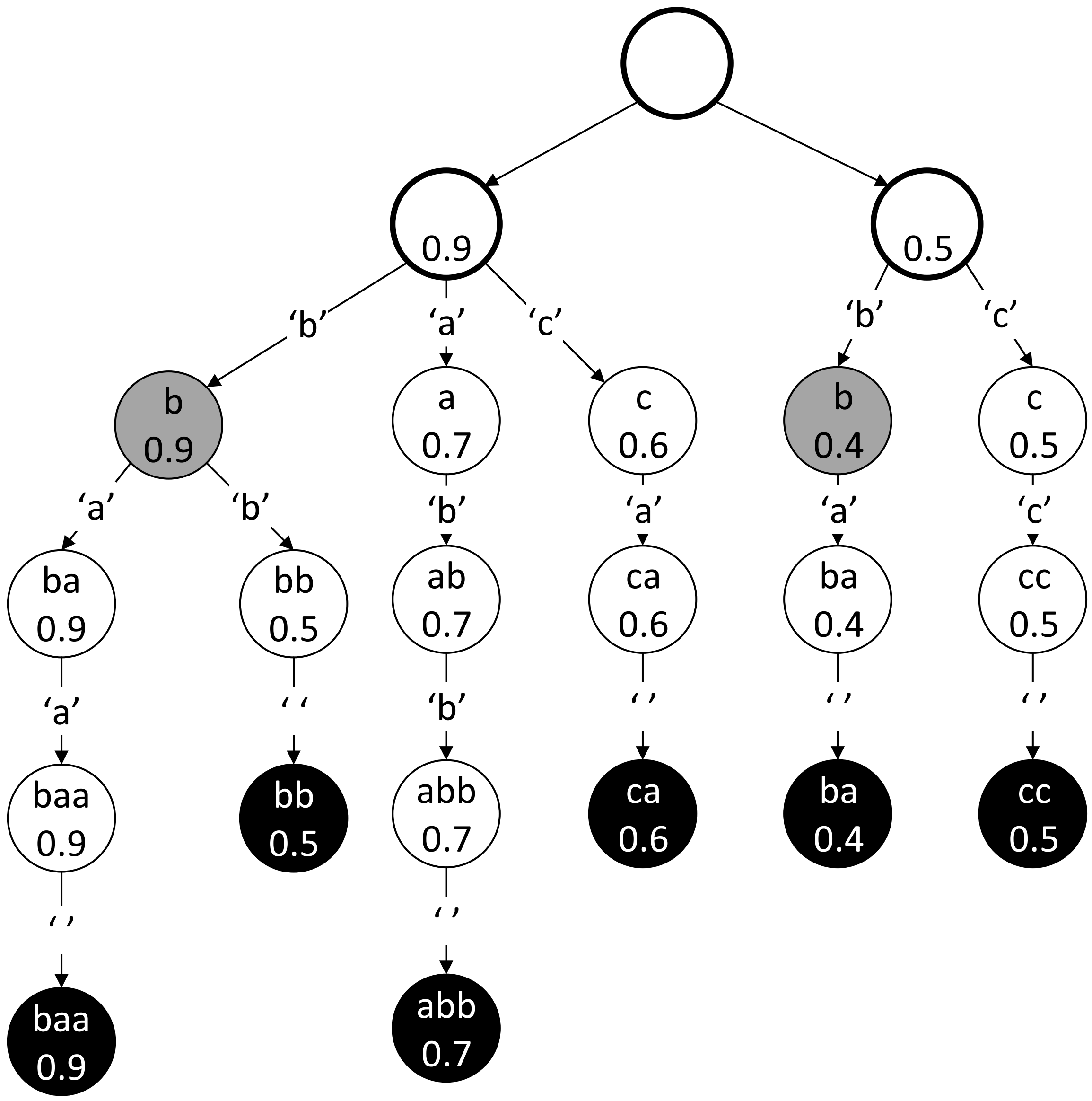


Figure 10: The active nodes resulting from the query "b" in a clustered trie.

When we have more than one active node, as will frequently be the case for fuzzy matching as well, the ability to retrieve suggestions one by one in order from each active nodes because essential. We need to check both the active nodes for their next suggestion rank before actually retrieving them from the node with the highest next rank. In this way we are able to retrieve them in the correct order by employing the algorithm from Figure 6.

3.5 INSERTING TERMS INTO A CLUSTERED SORTED TRIE

Inserting a term into the clustered sorted trie index becomes a much more complicated operating than in the non-clustered version. There is not only the problem of determining what cluster to put the new cluster in, but also the challenge of determining how many clusters there should be and the problem that the addition of a new term might affect all the clusters by skewing what terms belong to which cluster. Given the last effect, the addition of one term might in worst case cause a re-indexing of everything in the index. Even if this is a rare event, in is clearly undesirable for large indexes.

Solving these problems are beyond the scope of this thesis. Our approach assumes that we know at indexing time both the number of clusters and all the terms. This approach is thus only appropriate for a fairly static corpus. Also, our interest is to look at the performance characteristics of this arrangement compared to calculating term correlation directly. The quality of the clusters and, in turn, the quality of the multi-term suggestions are therefore also outside our scope.

To cluster the terms we will employ k-means clustering [24]. We will represent each term by with an inverted index, where each document is represented with the tf-idf weight for that term. For each of the finished clusters, we will then employ the algorithm from Figure 4 to build up each trie index. We will not try to optimize the numbers of clusters to build to improve the quality of the suggestions, because this is outside the scope of this thesis. We will, however, measure how the speed of a query scales with an increasing number of clusters.

4 Single Term Interactive fuzzy search algorithm

This section will introduce three different algorithms for interactively fuzzy matching a query with the trie index developed in chapter 0. The first algorithm is a simple baseline similar to the base algorithm used by Ji, et al. (2009) [7], Wang, et al. (2010) [17], Li, Ji, et al. (2011) [18] and Li, Wang, et al. (2012). The second and third are two different approaches to exploit the augmented trie structure to efficiently extract the top-k suggestions. First, we do however need to define how to do fuzzy matching, so we will look at the notion of edit operations.

4.1 Edit operations

To allow for fuzzy matches, we will use a set of standard edit operations [9]. We have the following operations in respect to a position in a trie and the query string.

- **Match.** Travels an edge from the current node where the label of the edge matches the corresponding character in the query string.
- **Insert.** Travels any of the edges from the current node while in effect inserting the label of that edge into the query string. The character that is to be evaluation in the destination node is therefore the same as in the source node.
- **Delete.** The effect of the delete operation is to delete the character from the query string evaluated at that node. The next step will then be to evaluate the next character from the query string (if any) at the same node in the trie.
- **Substitution:** Substitution is to exchange the character from the query string under evaluation with any of the labels of the available edges. In effect this is the combination of an insert and a delete operation.

The minimum number of edit operations (this excludes the match operation) needed to transform one word into another is called the minimum edit distance or the lev*enshtein distance* between these words.

To avoid duplicate results, we will apply some restrictions to the use of these operations. First, we will not use substitutions. We will rather allow a delete of cost 0 after an insert, which semantically gives an equivalent result. The advantage here is that this avoids the duplicate paths {insert + delete} and {substitution} where the former always will be worse than the latter due to using one more edit.

Further an insert operation will not be allowed to follow a delete unless that delete represented a substitution operation (came after an insert). This is because {delete + insert} is equivalent to {insert, delete} so we need only evaluate the latter.

The last constraint we set on edit operation usage is that it is not allowed to insert with a character that is also a match, since this always will be worse than just taking the match. The insert operation (if necessary) will instead be handled on the next node. Since the characters are identical inserting before or after will be equivalent, so it is safe to disallow the former.

4.2 Single term fuzzy ranking

In section 3.1 we added a rank to each term. This rank will be the basis when we rank suggestions in single term interactive fuzzy search. The ranking function we will use will be the following:

$$rank(S, Q) = termRank(S) \times editDiscount(S, Q)$$

Where Q is the query string and S is the Suggestion. The edit discount is a discount of the rank according to the number of edits we had to perform to match the query string to the retrieved suggestion. Because it makes little sense to have edit operation have a positive impact on the final score, we will dictate that:

$$0 \leq editDiscount(S, Q) \leq 1$$

This is also an important property because it means that the rank of a term will be an optimistic estimate of the rank that term would have as a suggestion for a query.

In our implementation we will use the following definition of edit discount:

$$editDiscount(S, Q) = \prod_{minEditDistance(S,Q)} C$$

C is here a constant between 0 and 1. In our examples and in our tests, we will use C = 0.5, but this constant can be tuned to regulate how much emphasis we give to the exact match between the query and a suggestion. Such tuning is however outside the scope of this study.

4.3 Common algorithm framework

The exact problem we are trying to solve here consists of two parts. First, given an incomplete single term query, find the top c (c < k) fuzzy matched prefixes that are within an edit distance of the maximal allowed edit distance. Then, we want to retrieve the k best suggestion given these c prefixes.

The three algorithms presented here fits into a common framework and will be used in the way described by the pseudo-code in Figure 11. The getSuggestions() function will be called each time the user types a new character. It iterate over a query object until it has retrieved k suggestions or there are no more of the index to explore. Each iteration will explore the next unit of the index and this can expose new suggestions. We will refer to the full treatment of a character as a character iteration, not to be confused with the iterations that are done on the query object. The query object may internally represent any of the three algorithms that we present bellow.

```
getSuggestions(query, k)
    suggestions = { }
    while query.hasMoreNodes AND |suggestions| < k
        query.exporeNextNodes()
        while query.hasAvailableSuggestions() AND |suggestions| < k
            suggestions.add(query.getNextAvailableSuggestions())
    return suggestions
```

Figure 11: Common usage of the interactive fuzzy search algorithms.

Each of the algorithms will use a query context object. This object contains constant values like the number of edit operation allowed and the root node of the index, but also grants access to variable objects, like the query string.

4.4 THE NAÏVE BASELINE ALGORITHM

As a baseline for this study, we will use the simple/naïve technique of generating every fuzzy match prefix within the maximal edit distance, then extract every term that has these partial terms as prefixes.

```
persistent:     queryContext
                activeNodes = { }
                suggesitons = { }
                suggestionPosition = 0
init( queryContext )
        queryContext = queryContext
        activeNodes.add( new ActiveNode( queryContext,  queryContext.IndexRoot, 0, 0 ) )

updateQueryString( char nextCharacter )
        queryContext.QueryString.addCharacter( nextCharacter )
        suggestions.clear()
        suggestionPostition = 0

exploreNextNodes()
        nextActiveNodes = { }
        foreach activeNode in activeNodes
                activeNode.getExhaustedDecendantNodes( nextActiveNodes )
        foreach exhaustedNode in nextActiveNodes
                exhaustedNode.getAllSuggestions( suggestions )
        activeNodes = nextActiveNodes

getNextAvailableSuggestion()
        suggestion = suggestions[suggestionPosition]
        suggestionPosition += 1
        return suggestion
```

Figure 12: The baseline algorithm

This algorithm will not make any use of the sorted nature of our index, and will therefore have to retrieve all suggestions for every prefix it finds. Pseudo-code for this is given in Figure 12 and Figure 13. The definition of the function addMatch() has been omitted because of its relative simplicity. The definition allowedToInsert() and allowedToDelete() is omitted to avoid cluttering the code with the state necessary to make these functions correct. Their function is to return true if the constraint mentioned about edit operations in 4.1 are met and that the resulting edit distance is less or equal to the maximal allowed edit distance.

```
persistent:    queryContext
               indexPosition
               editsPerformed
               queryStringPosition

init( queryContext, indexPosition, editsPerformed, queryStringPosition )
    queryContext = queryContext
    indexPosition = indexPosition
    editPerformed = editPerformed
    queryStringPosition = queryStringPosition

getExhaustedDescendantNodes( nextActiveNodes )
    foreach childTrieNode in indexPosition.Children
        candidatePathCharacter = childTrieNode.Label
        queryCharaceter = queryContext.QueryString.chatacterAt( queryPosition )
        if candidatePathCharaceter == queryCharacter
            addMatch( nextActiveNodes, childTrieNode )
        else
            addInsert( nextActiveNodes, childTrieNode )
    addDelete( nextActiveNodes )

addDelete( nextActiveNodes )
    if allowedToDelete()
        editCost = isDeleteSubstitution()
        deleteActiveNode = new ActiveNode(
                                        queryContext,
                                        indexPosition,
                                        editsPerformed + editCost,
                                        queryStringPosition + 1 )
        nextActiveNodes.add( deleteActiveNode )

addInsert( nextActiveNodes, trieNode )
    if allowedToInsert()
        insertActiveNode = new ActiveNode(
                                        queryContext,
                                        trieNode,
                                        editsPerformed + 1,
                                        queryStringPosition )
        insertActiveNode.getExhaustedDescendant( nextActiveNodes )
```

Figure 13: Pseudo-code for getExhaustedDecendantNodes

The algorithm is initiated with an active node for the index root. Active nodes represents positions in the index and the position in the query string that should be used to match when the active node are asked to explore more of the index. We will refer to an active node as exhausted if the position in the query string is the next position to be typed. We can extract suggestions from such an active node.

After the initiation the algorithm will do an iteration for each character that is typed and sent to it via the updateQueryString() function. Each iteration starts with the set of active nodes from the previous iteration. The exploreNextNodes () function extracts the child active nodes from this set of nodes via the getExhaustedDescendantNodes () function. All the resulting nodes are exhausted until the query string is further augmented. Note that in the getExhaustedDescendantNodes, we had to do a recursive call in the case of an insert, because the node resulting from an insert does not consume a query character.

After the next set of active nodes has been produced, all their suggestions will be extracted and put into the suggestion list. The list is then sorted so that the suggestions can be retrieved in order by the getNextAvailableSuggestion() function. For a top-k suggestions query, this function must be called k times.

The getExhaustedDecendantNodes() function in Figure 13 will add a match active node to the queue if the current node matches the query character associated with that node. Else it will add an insert node, if allowed. It will also add a delete node if allowed.

4.5 Example: the simple algorithm

This example will use the index from Figure 3, the query “ba”, the maximal edit distance of 1 and edit discount of 0.5. Let p be the position in the index, r be the rank of the active node, δ be the number of edits used and o be the last operation performed (M for match, I for Insert, D for delete and S for substitution). We will represent exhausted nodes as white, nodes used to generate active nodes as grey, and dead nodes as black. The nodes will also be numbered and we will show the suggestion queue and node queue, which refers to that node and gives their current rank in that queue. Note that the rank on the node is always their best child active node. We will concentrate on finding the active nodes, since it is the essential bit of the algorithm, and leave out the process of extracting the suggestions.

4.5.1 Initial

Initially, only a single active node exists representing the root node of the index. Every single suggestion relevant to this iteration can therefore be extracted from this node. Figure 14 illustrates the initial state.

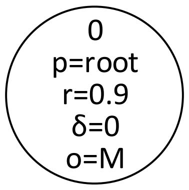


Figure 14: The initial state.

4.5.2 “b” iteration

When the user types b, we have four possible continuations from the root node (the only active node we have). We can match ‘b’, we can insert ‘a’, we can insert ‘c’ and we can delete ‘b’. All of these operations are legal, since our root node has δ=0. The insertion operations creates nodes that does not consume the query character, so these needs to be processed further. Neither inserts can continue with a match of ‘b’ since there are neither an “ab” or “cb” prefix in the index. No inserts are available, since they have δ=1. The only operation allowed is to convert them into substitution operations by using a free delete. The state after this iteration and the operations used to get there is illustrated in Figure 15.

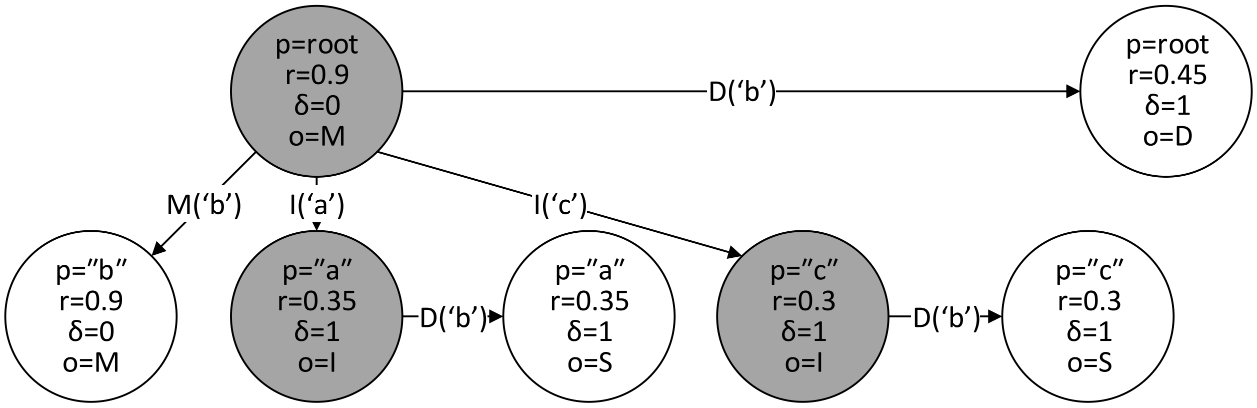


Figure 15: The state after the "b" iteration.

This iteration results in four exhausted nodes that may hold suggestions that we are interested in. They also have to be processed when the next query character is typed.

All the nodes that are marked as grey here are completely processed and can be discarded.

4.5.3 "ba" iteration

This iterations needs to iterate through all the four active nodes from the previous iteration. The result is displayed in Figure 16.

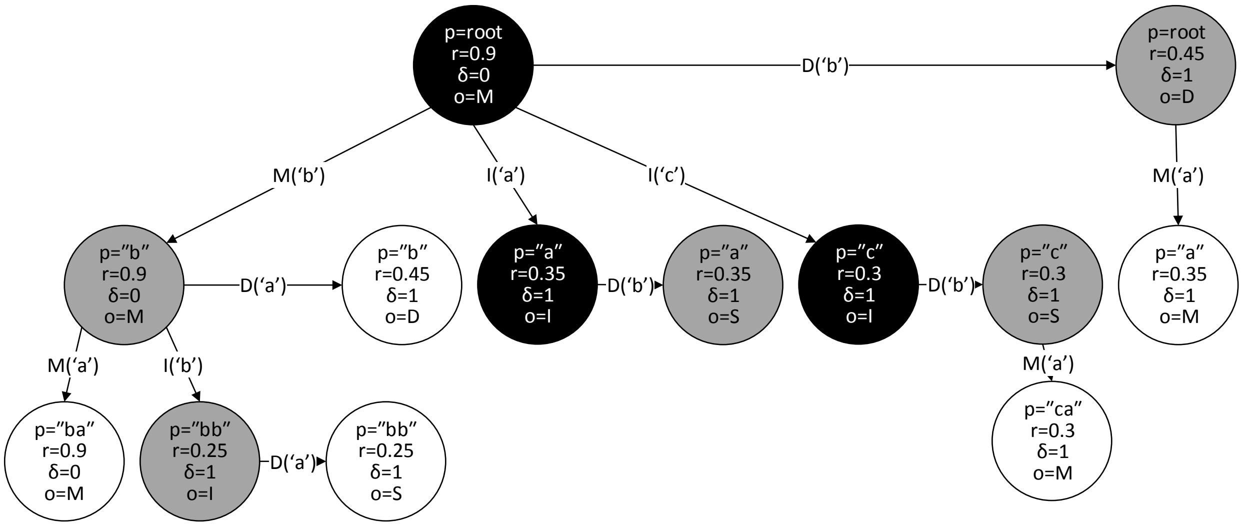


Figure 16: The state after the "ba" iteration.

Five exhausted nodes results from this iteration. We can see that because of the limited index size, many of the previous exhausted nodes were able to produce many new active nodes. Even so, this iteration is much more convoluted than the previous because of number of possible paths that needs to be checked. This is exactly the problem we will try to tackle with the two algorithms we will describe below.

4.6 THE QUERY STRING PRIORITY ALGORITHM

Because of the sorted trie index we developed in section 0, we have a way of knowing what nodes are most valuable when we make them. If we revisit Figure 16, we can see that the active nodes are ranked quite differently with “ba” with rank 0.9 as the maximum and “bb” for 0.25 as the minimum. If we could generate the best active nodes first and extract suggestions from them, we might be able to get all the suggestions we want without making the worst looking nodes.

The idea is to employ a technique similar to the branch and bounds technique, often employed to help solve NP complete problems [25]. As long as we can keep an optimistic (non-increasing) estimate of the value of every suggestion that descends from a particular active node, we can greedily explore the parts of the index with the highest rank before any other node. As long as we value an active node as the value of the underlying trie node multiplied by an edit discount value between 0 and 1, we can satisfy this condition.

To accomplish this in practice, we will keep a priority queue of active nodes based on their rank. It is initiated by adding an active node based on the root node, similar to the simple algorithm. Contrary to the simple algorithm, however, is that the unit of work is only to explore one of the nodes in the priority queue. A character iteration might therefore need to work through multiple such units before it has gathered all the k required suggestions.

In each unit of work, the best active node will be popped from the queue, and may be in two different states. It will either be exhausted (has consumed all of the currently available query string) or not. If it is exhausted, it will simply be put in a list of exhausted nodes and in a suggestion priority queue. If it is not exhausted it will be used to make children active nodes, similar to what we saw for the simple algorithm. However, instead of extracting all possible child, only the match child (if there is a match), the delete (if allowed) and the best insert will be added. The active node will keep track of what children it has created inserts for. If there are more inserts, the node will be put back into the priority queue, else, it will be discarded. The pseudo-code for the algorithm are given in Figure 17 and Figure 18.

```
persistent:     queryContext
                nodeQueue = new PriorityQueue()
                sugestionQueue = new PriorityQueue()
                exhaustedNodes = { }

init( queryContext )
        queryContext = queryContext
        exhaustedNodes.add( new PriorityActiveNode( queryContext, queryContext.IndexRoot, 0 ) )

updateQueryString(char nextCharacter)
        queryContext.QueryString.addCharacter( nextCharacter )
        foreach exhaustedNode in exhaustedNodes
                nodeQueue.add( exhaustedNode )
        exhaustedNodes.clear()
        suggestionQueue.clear()

peekNextNodeRank()
        if nodeQueue.size > 0
                return nodeQueue.peek().getRank()
        return  -1

exploreNextNode()
        currentNode = nodeQueue.pop()
        if currentNode.isExhausted()
                suggestionQueue.add( currentNode )
                exhaustedNodes.add( currentNode )
        else
                currentNode.getNextChildNodes( nodeQueue )
                if currentNode.hasMoreChildren()
                        nodeQueue.add( currentNode )

peekNextSuggestionRank()
        if suggestionQueue.size > 0
                return suggestionQueue.peek().peekNextSuggestionRank()
        return -2

getNextAvailableSuggestion()
        suggestionNode = suggestionQueue.pop()
        suggestion = suggestionNode.getNextSuggestion()
        if suggestionNode.hasMoreSuggestions
                suggestionQueue.add(suggestionNode)
        return suggestion
```

Figure 17: The query string based priority algorithm.

```
persistant:    queryContext
               indexPosition
               editsPerformed
               nextChildIndex = 0

init( queryContext, indexPosition, editsPerformed )
    queryContext = queryContext
    indexPosition = indexPosition
    editsPerformed = editsPerformed

getNextChildNodes( nodeQueue )
    if nextChildIndex == 0
        addMatchToQueue( nodeQueue )
        addDeleteToQueue( nodeQueue )
    addNextInsertToQueue( nodeQueue )

addDeleteToQueue( nodeQueue )
    if allowedToDelete()
        editCost = isDeleteSubstitution()
        deleteActiveNode = new PriorityActiveNode(
                                   queryContext,
                                   indexPosition,
                                   editsPerformed + editCost )
        nodeQueue.add( deleteActiveNode )

addNextInsertToQueue( nodeQueue )
    if allowedToInsert()
        if nextChildIndex < indexPosition.getNumberOfChildren()
            if isMatch( nextChildIndex )
                nextChildIndex += 1
        if nextChildIndex < indexPosition.numberOfChildren
            bextInsertNode = indexPosition.getSortedChild( nextChildIndex )
            nextChildIndex += 1

            insertActiveNode = new PriorityActiveNode(
                                   queryContext,
                                   bestInsertNode,
                                   editsPerformed + 1 )
            nodeQueue.add( insertActiveNode )
    else
        nextChildIndex  = indexPosition.numberOfChildren
```

Figure 18: Pseudo-code for a priority active node.

As for the simple algorithm, we omit the addMatchToQueue(), allowedToDelete() and allowedToInsert()

functions here because of their relative simplicity and to avoid convoluting the code with the state necessary to make the code for these functions correct. We have also omitted the isMatch(), which checks if a given trie node object will match the query string character associated with the active node, to avoid including further cluttering state variables.

4.7 EXAMPLE: QUERY BASED PRIORITY ALGORITHM

We will now look at the same example as in the previous example, but with our new approach. Further, in order to demonstrate the power of the technique used in this algorithm, we will only require two suggestions (the number of required suggestion does not matter for the simple algorithm). Note that grey nodes in the previous example would always get discarded in the next iteration. This is not the case when using the priority algorithm because a node may delay producing their last children nodes until they are actually needed, which might be several character iterations later, or in the best case not at all. The grey nodes are therefore nodes that are not exhausted and can produce more children nodes.

4.7.1 Initial state

There are nothing really new about the initial state. We essentially get the root node as the only active node.

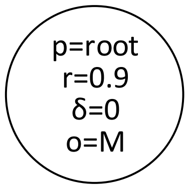
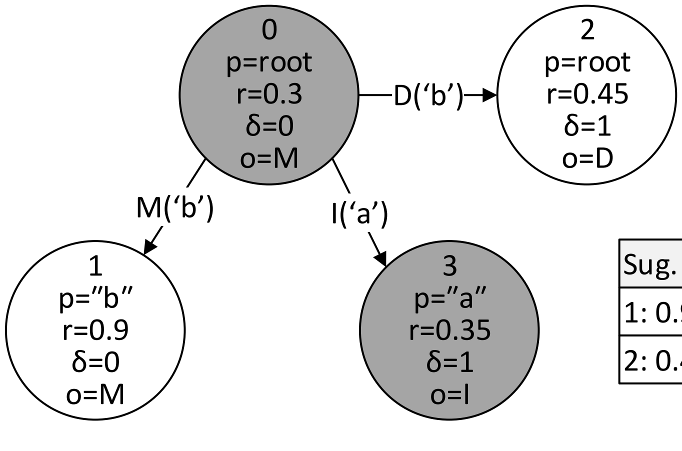


Figure 19: The initial state

4.7.2 "b" iteration

If we follow the algorithm in Figure 17 and explore one unit of the index, we arrive at the situation in Figure 20.



Sug. queue
1: 0.9
2: 0.45

Node queue
3: 0.35
0: 0.3

Figure 20: The state after exploring one unit of the index in the "b" iteration.

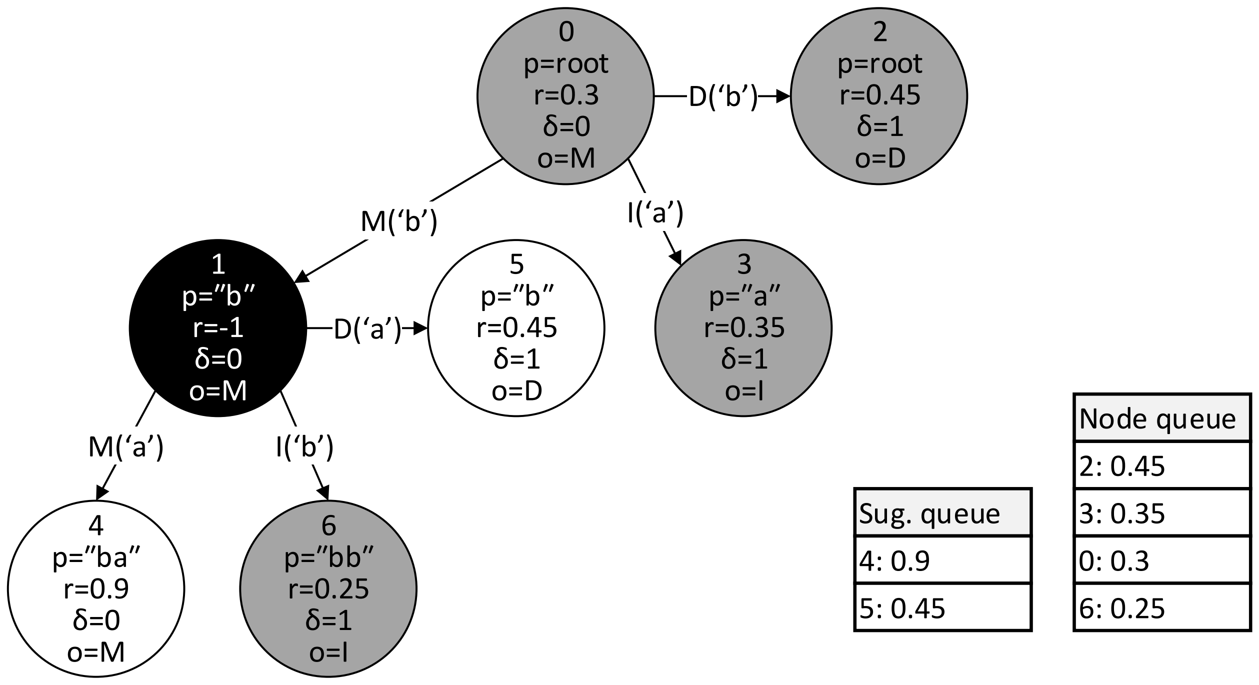
We see that we got the match, delete and the best insert node from the root node. Note that the rank of the root node changes to reflect that its next child has the rank of 0.3. Only the grey nodes, node 0 and

1, are actually in the node queue at this point. But we also have two nodes in the suggestion queue that are better than either of the grey nodes. If we use the functions defined in Figure 6 on the best, node 1 which represents “b” with rank 0.9, we can get both {“baa”, 0.9} and {“bb”, 0.5} witch still are above the rank of the next active node. Since we are only interested in the best two suggestions, this concludes this character iteration.

If we compare the state of Figure 20 and Figure 15, it’s obvious that we managed to avoid doing a lot of work. In particular while the simple algorithm produced 7 nodes, the priority algorithm used only 4. Further, if we needed more suggestions, we could easily just continue the algorithm.

4.7.3 “ba” iteration

This iteration starts of with all the nodes of Figure 20 in the nodeQueue. We start by popping the best node from the node queue, which is the “b” node. It is not exhausted, so we extract try to extract children from it. The resulting state is shown in Figure 21.



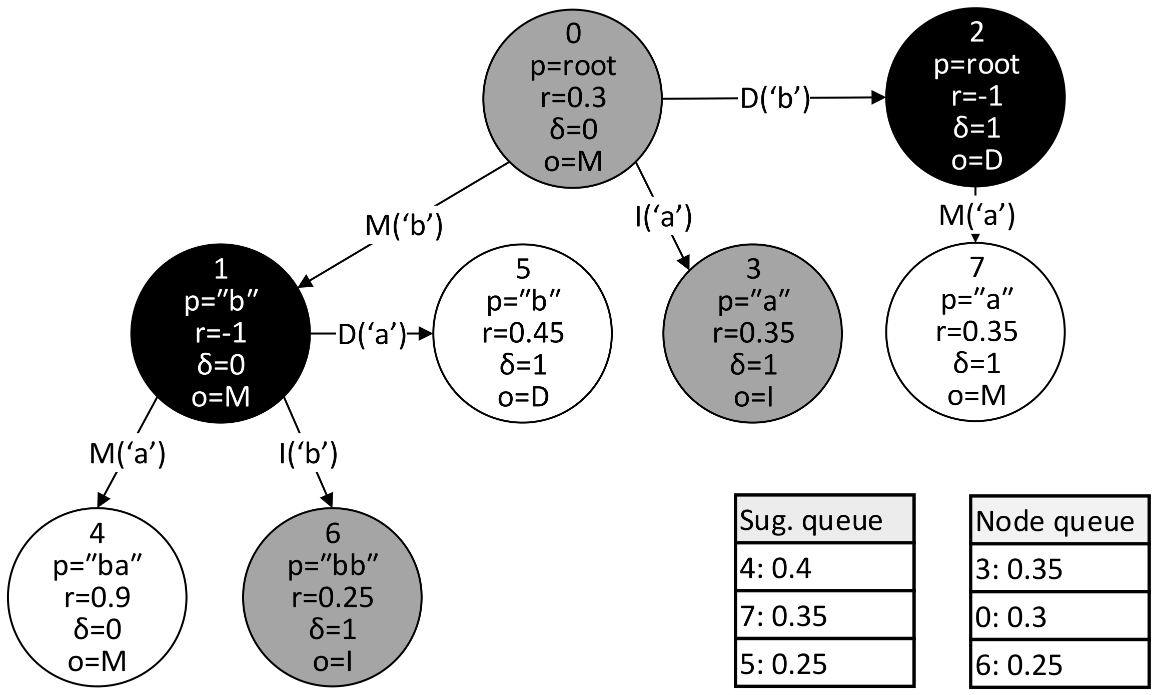
Sug. queue
4: 0.9
5: 0.45

Node queue
2: 0.45
3: 0.35
0: 0.3
6: 0.25

Figure 21: The state after expanding one unit of the index in the "ba" iteration.

From this state we have 4 nodes in the queue, one have been discarded and two nodes are in the exhausted nodes list and the suggestion queue. Since the best node in the suggestion queue are better than the best in the node queue, we try to extract suggestions. We get {“baa”, 0.9} from the “ba” node. The next suggestion from this node is now {“ba”, 0.4}, which is worse than what node 2 claims it can provide (0.45). So, the algorithm will continue to extract suggestions from “b”. The best suggestion “b” can provide is however {“baa”, 0.45}, a duplicate of the suggestion {“baa”, 0.9} which is already retrieved. The next suggestion “b” has to offer is only worth 0.25, so we will not try to retrieve it.

At this point, the node queue has the higher ranking with 0.45, so we need to explore another unit of the index. We get the root node resulting from a delete. Since it has δ=1 it can only generate a match child node. The result is given in Figure 22.



Sug. queue
4: 0.4
7: 0.35
5: 0.25

Node queue
3: 0.35
0: 0.3
6: 0.25

Figure 22: The state after exploring the second unit of the index in the "ba" iteration.

Now, the best node in the node queue is 0.35 while we have a possible suggestion with rank 0.4 in the suggestion queue. So we try to extract this suggestion and get node 4, which can retrieve {"ba", 0.4}. Since we now have two suggestions, the character iteration has concluded.

If we compare Figure 22 with Figure 16 we see that the simple algorithm produced 13 nodes while the priority algorithm only produced 8. One interesting detail is that in neither in this or the previous iterations were any of the nodes resulting from inserts even considered. So even if the index had been significantly larger, we might only have to explore the match node as we did in this example, while the simple algorithm would still have to explore every single possible insert from the root.

4.8 HANDLING SUGGESTION DUPLICATIONS

One detail we have somewhat glossed over in our discussion of the simple and the query string based priority algorithm is the possibility of duplicate suggestion. We briefly saw this situation when we tried to retrieve suggestions in the example above. If we study Figure 15, where we have produced all the possible nodes at that state, we see that we have one exhausted (white) node that has all the potential suggestions, namely the active node representing the index root. It is the result of a delete of the first typed character. Moreover we have a set of substitution nodes and a match node, which together also cover every single term in the index (except for a potential empty string), because they represent all possible single character prefix.

In this case we would be able to get away with just ignoring the active node representing the index root and extract suggestions from every match and substitution node. Thing can, however, get significantly more convoluted once we continue from the delete nodes and if the allowed edit distance is greater than 1. Thus, making rules for when to extract suggestions from which nodes to avoid duplicates are infeasible in this framework and would at least require to store more state in the active nodes.

Duplicates could potentially easily be removed when they are extracted by simply checking the list of previously extracted suggestions. The problem with this solution is that we potentially would risk additional disk accesses if the index resided on a hard drive. There are also some implementation difficulties when the suggestions we try to retrieve seems to have a higher rank because there is a duplicate "in front" of the next suggestion we are really interested in. We have therefore instead devised a suggestion register that keeps track of the nodes in the index that has been traversed. This is a slight modification to the algorithm from Figure 6.

4.9 THE INDEX PREFIX PRIORITY ALGORITHM

In the previous section we discussed the suggestion duplication problem inherently present in the two algorithms presented above. In this section we will present an algorithm that are based on the same best first principle as the query string based priority algorithm, but that does not produce duplicates.

The way duplicates arises is that the algorithms are based on modifying the query string to match some position in the trie index. Each active node in the two previous algorithms essentially represents a query string that are within the maximal edit distance of the original query string. Because deletions and inserts are allowed we may end up with pairs of active nodes where one represents a prefix of the other, thus one represents a position in the trie index that are an ancestor of the position the other active node represents.

4.9.1 Non-overlapping active node set.

Here we will define a non-overlapping active node set. Intuitively, this is a set of active nodes representing nodes in the trie that are guaranteed to generate exactly one of each suggestion relevant to the query string. The flowing criteria must be satisfied for a trie node to be represented in this set:

1. The prefix represented by the trie node must be within the allowed edit distance compared to the query string.
2. A trie node must have no descendant with a lower edit distance compared to the query string.
3. No pair of represented trie nodes can be in an ancestor/descendant relationship.

Criteria 3 guarantees no duplications as it forbids the existence of active nodes that represents a prefix of another active node. Two active nodes can therefore not be the ancestor of the same terms. Criteria 2 makes sure that the best active nodes are represented. Note that if an active node has a better ranked descendant it will never actually be needed for suggestions, since it must have done an edit operation in order to be lower ranked than its descendant, and its source active must therefore also have been able to make active nodes representing each of its children. This means that the existence of an active node that satisfy criteria 1 where a descendant is higher ranked guaranties that there exists descendant active nodes that covers all descendant terms of that active node, and that these terms will be at least as highly ranked form these descendant active node. A set satisfying criteria 1-3 would thus represent each relevant suggestion exactly once.

Two possible such sets for the query "b" on the index from Figure 3 with a maximal edit distance of 1 is shown in Figure 23 and Figure 24. The latter displays a larger set of nodes than the former, but it is still valid since "b" can be transformed into a prefix of "ba" and "bb" with 0 edit operations. We therefore also get the notion of a minimal overlapping active node set, which is the one displayed in Figure 23 in this case. Note that having only the root in the set would not be valid, because it would require us to delete ‘b’ and the suggestions with a prefix ‘b’ would then be lowered ranked than in the case of Figure 23 and Figure 24.

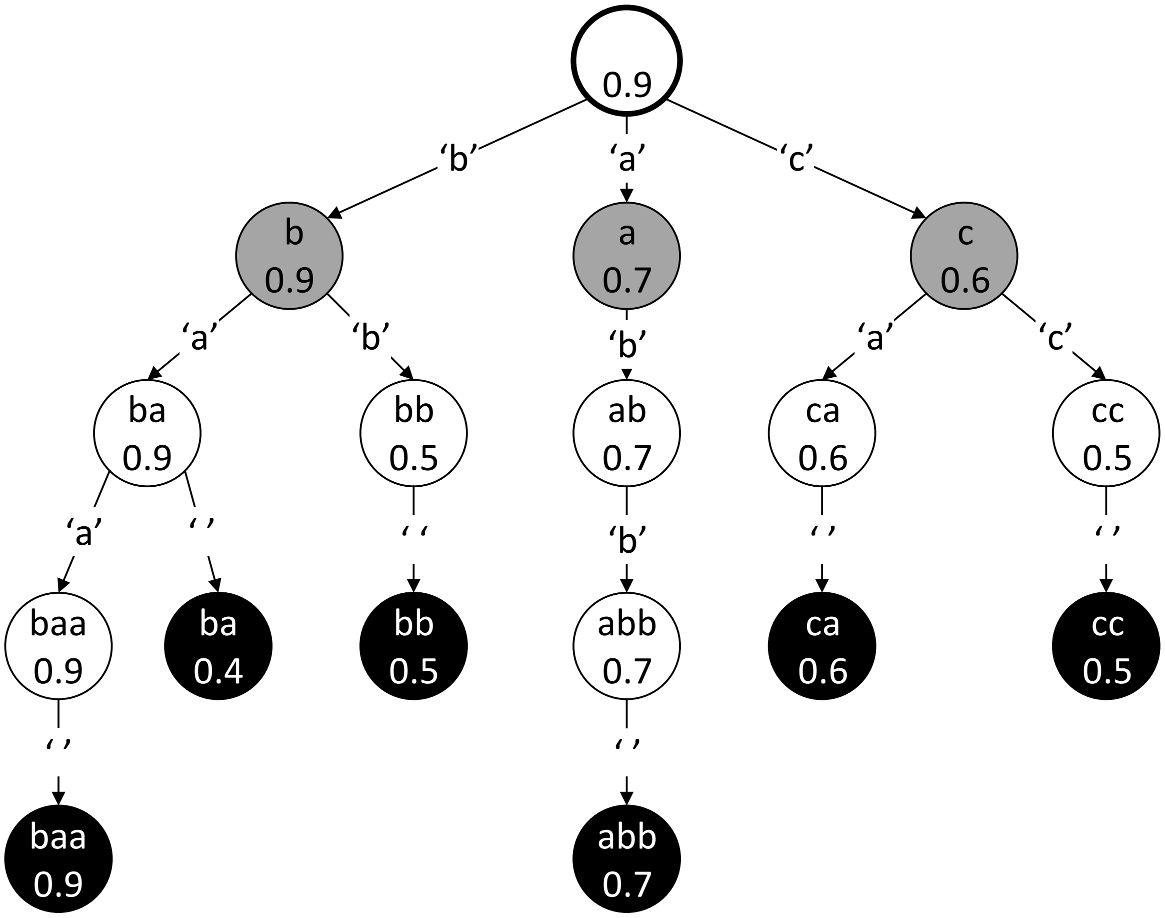


Figure 23: Minimal node set to cover all relevant nodes for the query "b".

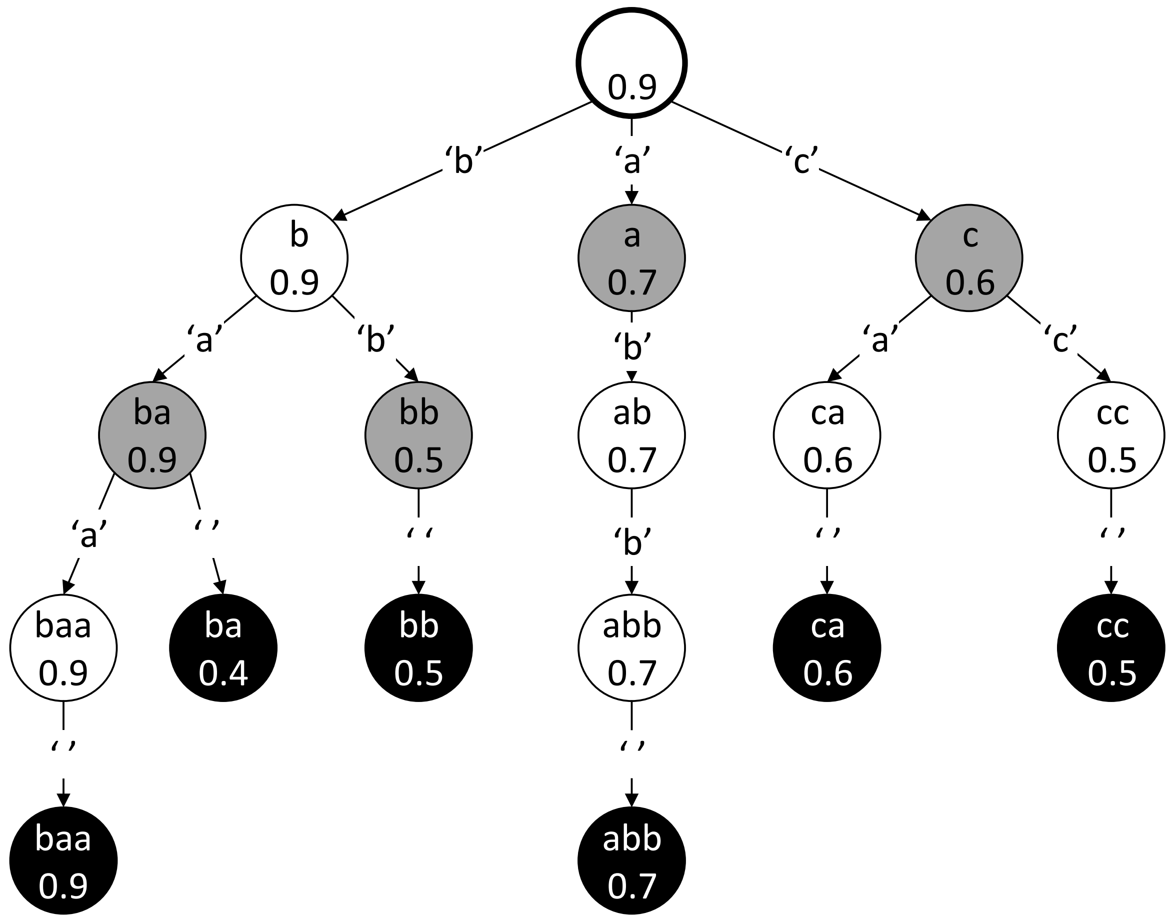


Figure 24: A set that covers the best ranked prefixes of all the terms relevant to the query "b".

4.9.2 Levenshtein distance and an efficient way to calculate it.

To find the rank of each active node, we will move away from trying different edit operations, and instead just calculate the minimum edit distance at an active node directly. We will employ a modified version of a dynamic programming approach to calculate the levenshtein distance. A comprehensive study of this algorithm is done by Ukkonen (1985) [9]. The basic algorithm is given in Figure 25.

```
getLevenshteinDistance( sourceString, targetString )
    distanceMatrix = new matrix( sourceString.length + 1, targetString.length + 1 )
    for i = 0 to sourceString.length
        distanceMatrix[i, 0] = i
    for i = 0 to targetString.length
        distanceMatrix[0, i] = i
    for i = 1 to targetString.length
        for j = 1 to sourceString.length
            if sourceString[j] == targetString[i]
                distanceMatrix[j, i] = distanceMatrix[ j – 1 ]
            else
                distanceMatrix[j, i] = 1 + min( distanceMatrix[j – 1, i],
                                                distanceMatrix[j, i – 1],
                                                distanceMatrix[j – 1, i – 1] )
    return distanceMatrix[sourceString.length – 1, targetString.length – 1]
```

Figure 25: A dynamic programming algorithm for computing the levenshtein distance between a source and a target string.

To illustrate how this algorithm works we will present an example of the edit distance between source "bab" and the target "baa" in Table 1.

	ε	b	a	a
ε	0	1	2	3
b	1	0	1	2
b	2	1	1	2
a	3	2	1	1

Table 1: The result of comparing "bab" to "baa".

The table displays the matrix that are built in the algorithm from Figure 25. Each position gives the minimum edit operations that are required to transform the source substring to the target substring. The string "ba" may be transformed into "b" with 1 edit operation, and "bb" may be transformed into "baa" om 2 edit operations. The minimum edit distance between the full strings are given by the cell in the lower right corner, in this case 1 (marked with black).

In our application, we will consider a match of the full query string (the source string) to some prefix of the target string. "bba" should therefore be considered a fuzzy match for "bac", shown in Table 2, because it can be transformed into "ba" with one edit, and "ba" is a prefix for "bac". This situation corresponds to a delete operation. We thus need to consider the entire bottom row for minimum edit distance.

	ε	b	a	c
ε	0	1	2	3
b	1	0	1	2
b	2	1	1	2
a	3	2	1	2

Table 2: The result of comparing "bab" to "baa".

Another case that may arise is a case where we could have a lower minimum edit distance if we consider a target string that is one step longer. If we compare “ro” to “pr” (Table 3), we need 2 edit operation to transform the former to the latter. Since “pr” are a prefix from the index, however, we could choose to instead compare “ro” to “pro”, which only requires 1 edit operation. This situation corresponds to an insert operation. In our application this means that if the last column (excluding the last position in the row) in a comparison has a lower value than the entire last row, we need to expand the target string (if possible) to check if there are a better match on the longer target string.

	ε	p	r	o
ε	0	1	2	3
r	1	1	1	2
o	2	2	2	1

Table 3: The result of comparing "ro" to "pr" and "pro".

If we look at the active node sets from Figure 23 and Figure 24, the latter would be a situation that would arise from this kind of case.

An important property of the algorithm to calculate Levenshtein distance is that we extend the source and target string, we need only the last row and column to calculate the next row and column. This is what makes this algorithm suitable for our needs because it makes us able iteratively calculate the edit distance as the query string grows and we move further down the index.

Another notable property is that we can avoid that the rows and columns grow each time we need to extend the source and target strings. Because we have a maximal allowed edit distance and because of the fundamental property of string distance that states that the edit distance between two strings is at least the difference in their length, we can limit the rows and columns to where the edit distances may be smaller than the maximal edit distance. For instance, if we have a maximal edit distance of 1 in the example from Table 1, we can drop everything marked with black in Table 4. We see, for instance, that the string “b” can never be transformed into “baa” within an edit distance of 1.

	ε	b	a	a
ε	0	1	2	3
b	1	0	1	2
b	2	1	1	2
a	3	2	1	1

Table 4: The needed parts (in white) of the distance matrix when calculating min edit distance with a maximal edit distance of 1.

Because of this property, we only need to keep the maximal edit distance + 1 numbers to calculate the next iteration of this iterative algorithm, which means that the space usage will be constant for each node for the duration of a query session. A corollary to this is that the search for the smallest edit distance in the last row and column can also be done in constant time.

An important detail here is that in an insert situation (Table 3), the needed amount of cells grows by one for the row, and shrinks by one for the column. Table 5 illustrates this case. Since we both gain and lose the need for a cell in the row and column respectively, the total number of needed cells are still constant.

	ε	p	r	o
ε	0	1	2	3
r	1	1	1	2
o	2	2	2	1

Table 5: The needed parts (in white) of the distance matric when doing an insert.

4.9.3 Algorithmic description

To produce the active nodes that are part of a non-overlapping active node set, we consider only nodes that are at a depth equal to the length of the query string or deeper in the index. Since we search the whole row for minimum edit, as we did in Table 2, we do not need to consider shallower depths in the index because we will essentially consider every prefix of the partial term that the node at the current depth represents. We may because of this end up with a set of nodes that are bigger than the theoretical minimum, but we will not be concerned with that in this study as it will not impact the correctness of the algorithm.

To find the best active nodes that are at the required depth, we do however employ the same technique that we used for the query string based algorithm. The algorithm is initiated with an active node representing the root, which is a valid set for the empty query string. Similar to the query string based priority algorithm we will keep a priority queue of the active nodes. For each character iteration, the algorithm may explore a new unit of the index by popping the best node from the node queue. If the node is exhausted, it will extract suggestions, or, if it is not exhausted, it will extract the next sorted child of the trie node. The minimum number of edit operation required to transform the query index to the partial term represented by this child is then calculated by employing the basic workings of the algorithm given in Figure 25. An active node representing the child will then be put into the node queue, along with the source node if it has more children.

Note that we cannot know which of the children will be highest ranked before we actually produce them. We could find a child node that matched the corresponding query string as a heuristic, but there may be other nodes that have no additions to the edit distance compared to the active node that spawned, as we saw in Table 3, when the column contained a cell with a lower edit distance than the last cell in the column. This case cannot be easily found by simply comparing the character for the node with the current query character.

There is a case that we need to take special care of and it is the case when the query string gets longer than some term in the index. This term might for instance be a prefix of the query string and might still be a fuzzy match. Our algorithmic description does however only move forward further down in the index, and would simply ignore these terminal nodes. To deal with them, a special node is made that keeps track of the length of the query string where its edit distance were computed. If this length matches the length of the query string when the node is taken from the queue, the node is used for suggestions, else, it will recalculate the edit distance and put the node back into the queue if it is still within the allowed edit distance.

The code from Figure 17 is still valid for this new algorithm and can be reused. Only the behavior of the nodes needs to be changed. Pseudo-code for the prefix based active nodes are given in Figure 26. The function handleLeafNode() handles the case of when we reach the terminal nodes as described above. If the column row has the lowest value, corresponding to the case in Table 3, the signalNeedMoreDepth() is run. This really a signal to the next produced node and will be used in the MakeNextPrefixActiveNode function. We have omitted the fields needed to make this work to avoid clutter in the code. If signelNeedMoreDepth() was ran in the parent node of the current node, needMoreExtraDepth() will be return true. In this situation the next active nodes min edit distance will be calculated as in Table 3, where we extend the depth in the index we compare with by one. The calculateMinEditDistanceWithExtensions() function handles this (the computation are a bit different, so it needs to be handled separately).

In the normal case we do not need extra depth, and the edit distance for the next node is calculated with the calculateMinEditDistance() function.

```
persistant:     queryContext
                indexPosition
                nextChildIndex = 0
                editRow
                editCol

init(queryContext, indexPosition, editRow, editCol)
    queryContext = queryContext
    indexPosition = indexPosition
    editRow = editRow
    editCol = editCol

getNextChildNodes( nodeQueue )
    nextChildTrieNode = indexPosition.getSortedChild( nextChildIndex )
    if isLeaf( nextChildTrieNode )
        handleLeafNode( nextChildNode )
    else
        nextChildIndex += 1
        nextCol = { }
        nextRow  = { }
        if needMoreExtraDepth()
            calculateMinEditDistanceWithExtension( nextCol, nextRow )
            signalOneExtraDepthAdded()
        else
            calculateMinEditDistance( nextCol, nextRow )
        nextEditDistance = getMinValue( nextRow )
        minColDistance = getMinValue( nextCol )
        if minColDistance < nextEditDistance
            nextEditDistance = minColDistance
            signalNeedForMoreExtraDepth()
        if nextEditDistance <= queryContext.MaxEditDistance
            nextActiveNode = MakePrefixActiveNode( queryContext, nextChildTrieNode
                                                   nextRow, nextCol )
            nodeQueue.add( nextActiveNode )
```

Figure 26: Pseudo-ode for a prefix based active node.

4.10 EXAMPLE: THE PREFIX BASED PRIORITY ALGORITHM

We will again revisit the example we used for the two previous algorithms. We will modify the illustrations somewhat, by adding last rows and columns

4.10.1 Initial state

The initial state here will represent the comparison between the empty query string and the empty index string. It requires 0 edit operations to transform the former to the latter, so the row and column (respectively) has the value 0.

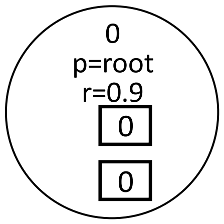


Figure 27: The initial active node.

4.10.2 "b" iteration

We explore the first unit of the index, and end up with the situation we see in Figure 28. Because the 'b' edge was a match for "b", which was a coincidence as the algorithm will explore the nodes in order of their rank without looking for matches first, both the row and the column for node 1 has 1 has the first value and 0 as the second. The minimum edit distance is therefore 0. Node 1 is also exhausted, so we can mine it for suggestions. We retrieve { "baa", 0.9 }. The next suggestion rank at this point is 0.5, which is less than the estimate for the value of the next child of node 0.

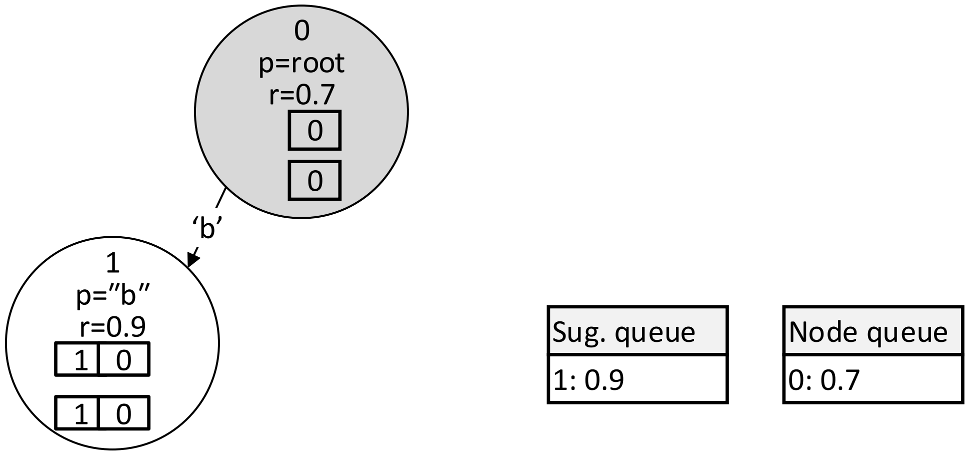


Figure 28: The state after exploring one unit of the index in the "b" iteration.

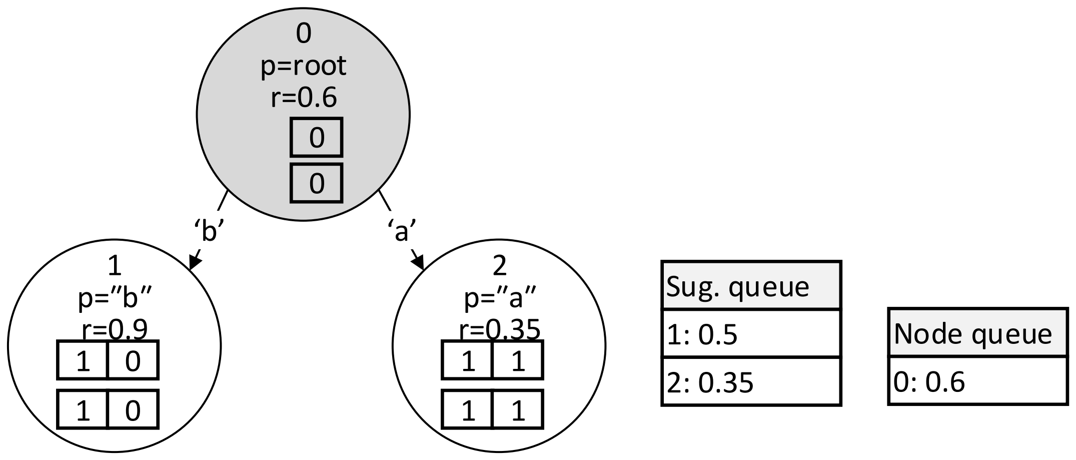


Figure 29: The state after extracting 2 units of the index at the "b" iteration.

In Figure 29 another unit of the index has been explored. This does not give any new suggestions as the best node in the node queue still has a higher rank than the next suggestion. We therefore explore another unit of the index from node 0 as Figure 30 shows. At this point we have produced every possible child from node 0, and together they represents the complete non-overlapping set of active nodes for this character iteration. We can now retrieve the second suggestion, {"bb", 0.5}, which concludes the character iteration.

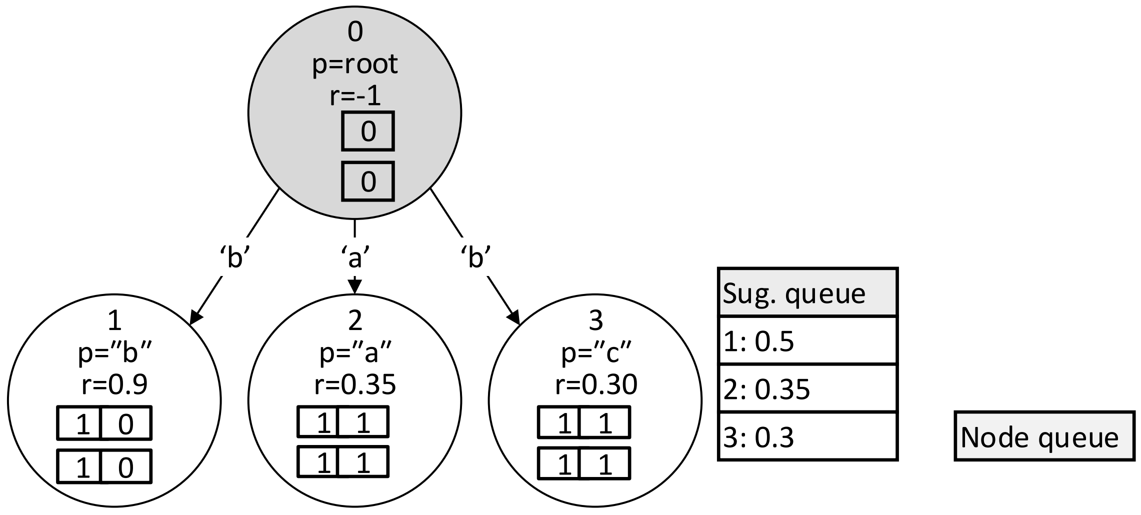


Figure 30: The state after extracting 3 units of the index at the "b" iteration.

4.10.3 "ba" iteration

This iteration starts by extracting another unit of the index, and we get the "ba" node, as seen in Figure 31.

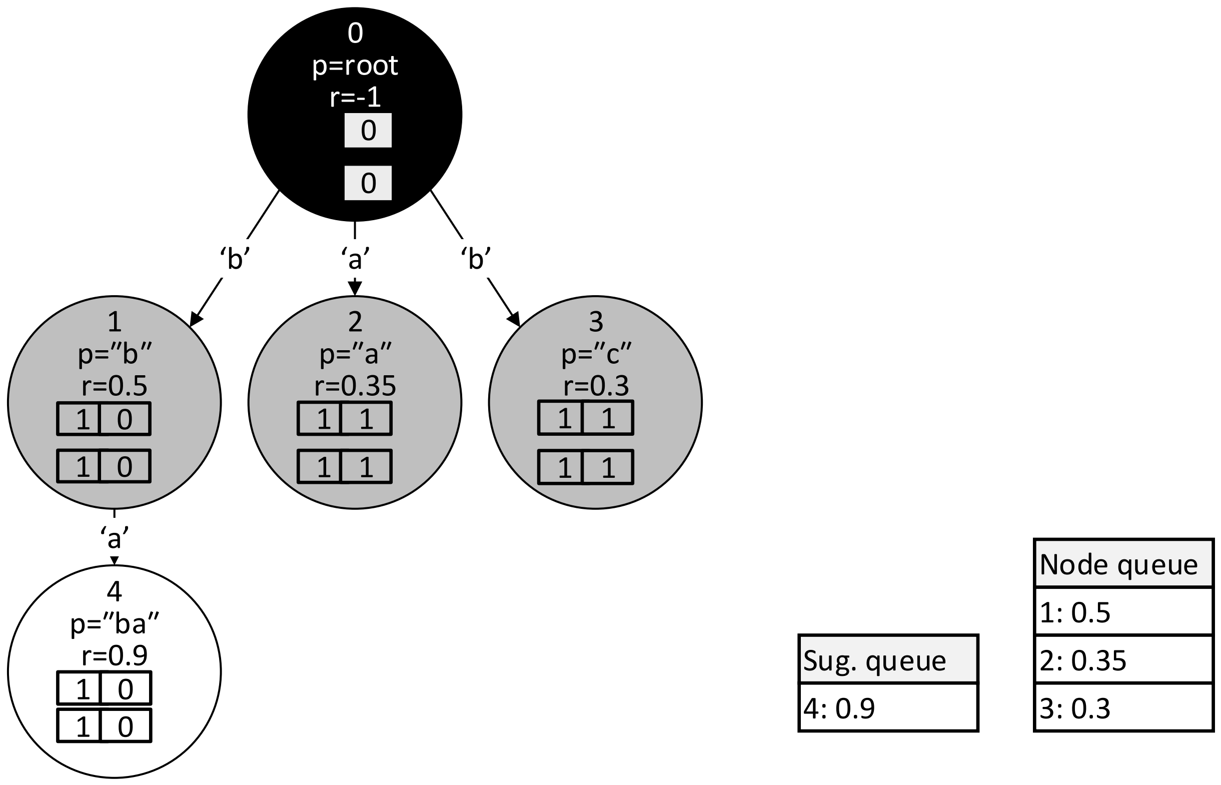


Figure 31: The state after expanding one unit of the index in the "ba" iteration.

Again, we can retrieve the suggestion {"baa", 0.9}. The next suggestion has a rank of 0.4, so we have to explore more children from node 1. The resulting "bb" node shown in Figure 32 are ranked lower than the suggestion available from node 4 so we can retrieve the second suggestion {"ba", 0.4} form node 2.

In this character iteration we only had to make two nodes, and ended up at a total of 6 nodes, which is one less than the query string based priority algorithm made. We also completely avoided duplicate suggestions. The nodes are however more expensive to make. Moreover, the matching character happened to also be the highest ranked. If it had been of lower rank, we could possibly spent a lot more time finding good nodes because we cannot predict a nodes rank before we calculate its minimum edit distance.

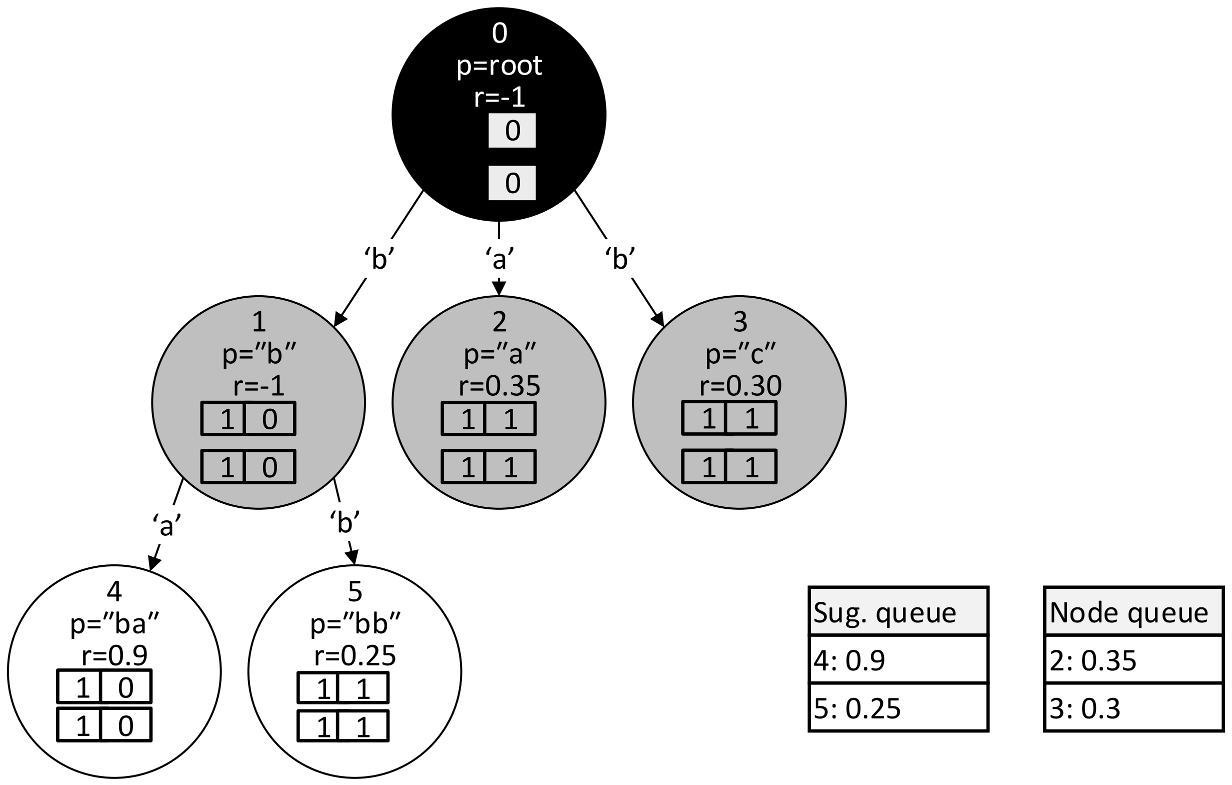


Figure 32: The state after exploring two units of the index in the "ba" iteration.

4.11 CAVEATS: MODIFYING THE QUERY STRING IN OTHER WAYS THEN APPEND

While the algorithms presented in this section has the potential to reduce the need to modify the typed parts of the query string, this is still an operation that potentially can be useful. The exploratory nature of interactive search does encourage the user to try typing in letters in a search box without prior knowledge of exactly what the query should look like. In this use case it is important to be able to backtrack. Neither of the algorithms presented here do however support this in any other way than just restarting. They have no inherent way of rolling back operations or trying to assess how to continue after a modification of the query string other than an append operation.

The way these algorithms are able to keep interactive speed is their iterative nature. Recalculating the entire query from scratch for long query string might cause a noticeable time gap before the user receives the first suggestions.

A way to avoid this problem is to regularly save the state of the query as a checkpoint and roll back to the closest checkpoint that is still valid. This would require more memory and the states may already be quite large. These checkpoints would also only be able to help up to a point. If the modifications to the query string is at the start, there will still be a lot of work to do because an early checkpoint must be used.

5 Multi term interactive fuzzy search algorithm

In this section we will augment the single term algorithms we presented in the previous section to support multi term queries. We will look at three different approaches involving different matching approaches and ranking schemes. Unlike the single term case, the algorithms presented here will generally not return the same suggestions, but we will define ranking function based on the same concept, which at least should not make the difference too large.

5.1 Matching multiple terms as one large string

Terms in a multi term query are delimited by the space character in natural language. One way to match a multi term query is to simply match the space character as a regular character. We enabled this approach by using the space character as the termination character in the index we developed in section Figure 2. The resulting node is a special termination node. If we detect that the matched node is a termination node, we could interchange it with the root node and save the full term that was matched. The query could then just continue through the index a second time. An example of this traversal method is given in Figure 33 where a query sting “baa abb” gives an exact match in the index showed in Figure 3.

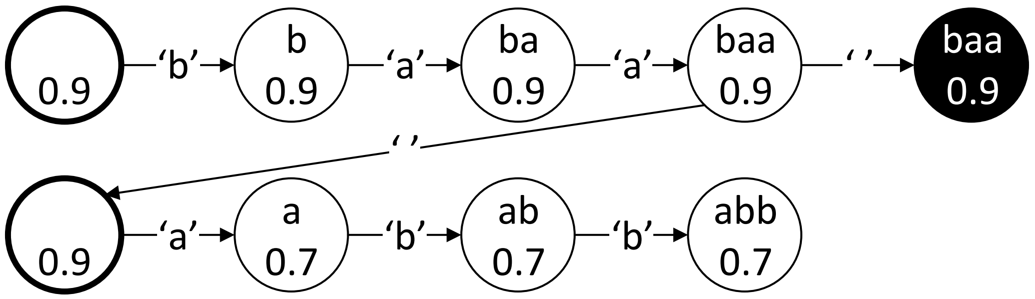


Figure 33: Matching "baa abb".

One of the advantages with this method is that the space character can be fuzzy matched like a regular character. We will therefore be able to retrieve the same suggestion “baa abb” even if we accidently left out the space character and typed “baaabb”. We will also be able to delete an accidently inserted space and still get “baa abb” if the query was “b aa abb”.

Another advantage with this method is that it is a relatively simple extension over the single term algorithms. The only thing required is to treat the leaf nodes specially by replacing them with the root node and, in that process, keeping track of the previously matched terms. All three algorithms can be adapted to support this in almost the same way.

One disadvantage of this method is that the matched string can become very long. Since the number of nodes may increase exponentially with the length of the matched string, this could potentially be slow. The need to “guess” the correct nodes early and avoid visiting nodes that are unlikely to result in a suggestion will therefore be very important. The simple algorithm, which does not do anything to avoid visiting nodes is therefore likely to have significant speed issues with this approach.

Another challenge is how ranking should be done for the matched combined multi term string. This challenge arise from the property that the ranking needs to be non-increasing for the two priority algorithms to work correctly. The combined rank for two terms must therefore at best stay the same even if the second term has a higher rank than the first. This limits our options when choosing a ranking function. Note that it is also important that the rank does not decrease too much, since it will be compared with other active nodes that has had its rank discounted as a result of an edit operation. While some of these nodes may be interesting, most will not, and as we stated above, it is really desirable to avoid visiting nodes to avoid slowdowns.

```
persistent:     previousTerms

init( previousTerms )
    previousTerms = previousTerms

addNextInsertToQueue( nodeQueue )
    if allowedToInsert()
        bestInsertNode = null
        if nextChildIndex < indexPosition.getNumberOfChildren()
            if isMatch( nextChildIndex )
                nextChildIndex += 1
        if nextChildIndex < indexPosition.numberOfChildren
            bextInsertNode = indexPosition.getSortedChild( nextChildIndex )
            nextChildIndex += 1
        insertActiveNode = null
        if isLeafNode( bestInsertNode )
            newPreviousTerms = { }
            foreach term in previousTerms
                newPreviousTerms.add( term )
                newPreviousTerms.add( bestInsertNode )
            insertActiveNode = new PriorityActiveNode(
                                        queryContext,
                                        queryContext.IndexRoot,
                                        editsPerformed + 1
                                        newPreviousTerms )
        else
            insertActiveNode = new PriorityActiveNode(
                                        queryContext,
                                        bestInsertNode,
                                        editsPerformed + 1,
                                        previousTerms )
        nodeQueue.add( insertActiveNode )
    else
        nextChildIndex  = indexPosition.numberOfChildren
```

Figure 34: Extension of addNextInsertToQueue that supports multi term.

The required extension to the addNextInsertNode function from the query string based priority algorithm is given in Figure 34. The code checks if the next node from the index is a leaf node and switches it for the root node if it is. It also puts the node into a new previousTerms structure, which it passes to this next node. We have omitted the list of global members here for simplicity, but we assume that it contains the same members as the single term case.

Figure 35 gives the required extension to the suggestion retrieval algorithm. It will pass forward the previous terms list, so that these can be put in front of the retrieved terms.

```
persistent:     suggestionQueue = new PriorityQueue()
                previousTerms

init(trieNode, previous terms)
        suggestionQueue.add(trieNode)
        previousTerms = previousTerms

getNextSuggestion()
            while suggestionQueue.size > 0
                    currentPosition = suggestionQueue.pop()
                    if currentPosition is Leaf
                            return makeMultiTermSuggestion( currentPostition, previousTerms )
                    else
                            nextChild = currentPosition.getNextChild()
                            suggestionQueue.add(nextChild)
                            if currentPosition.hasMoreChildren()
                                    suggestionQueue.add(currentPosition)
            return null

getNextSuggestionRank()
        if suggestionQueue.size > 0
                currentNode = suggestionQueue.peek()
```

Figure 35: Multi term suggestion retrieval.

5.2 Calculating term correlation

We have now seen how we can extend the single term algorithms to support multi term queries by matching them as a large string. We will now develop two ways of calculating a rank for multi term suggestions.

5.2.1 General multi term ranking

Because we depend on the branch and bounds technique to enable the priority algorithms to be correct, the ranks must always be non-increasing. This means that when we combine the rankings of two individual terms, we cannot increase the rank even if the latter term has a higher rank than the former. Given suggestion S = $\{s_1, s_2 \dots , s_n\}$ we will use the following ranking function:

$$termsRank(S) = \prod_{i=1}^{n} termRank(s_i)$$

To make this function non-increasing as the number of individual terms increase, we normalize all the ranking of each individual terms in the range of [0, 1].

$$0 \leq termRank(s) \leq 1$$

The termsRank measure only depends on the rank of the individual terms. To help the user choose a sound direction to continue the query, we would also like to take semantic correlation between the individual terms into account. This will also be a value in the range [0, 1].

$$0 \leq corrRank(s) \leq 1$$

The complete ranking function for a suggestion S then becomes:

$$rank(S, Q) = termRank(S) \times corrRank(S) \times editDiscount(S, Q)$$

We will present two ways of calculating the correlation rank. The first is a direct calculating that is performed when all the terms are found. The second is based upon the clustered index we presented in section 3.4. We will later refer to the former approach as the direct long string algorithm and the latter as the clustered long string algorithm.

5.2.2 Direct semantic correlation calculation

We will use the notion of the cluster center of the partial suggestion to calculate the semantic correlation rank. We use this approach to have an iteratively calculable score that is similar to the score we get when using clusters. The only difference is that the cluster score will be cruder. The code in Figure 36 outlines the function to calculate the corrRank that we have used.

```
calculateSemanticCorrelation(termNodeList)
    correlation = 1.0
    **if** termNodeList.size != 1
        clusterCenter = new ClusterCenter()
        clusterCenter.add(termNodeList[0].documentPosting)
        **for** i = 1 **to** termNodeList.size – 1
            termNode = termNodeList[i]
            distance = clusterCenter.getDistance( termNode.documentPosting )
            correlation *= getDistanceScore( distance )
            clusterCenter.add( termNode.documentPositng )
    **return** correlation
```

Figure 36: Calculating semantic correlation directly.

The getDistanceScore function gets a score in the range between 0 and 1 from the distance that the i'th term are from the centroid of the i – 1 previous terms. To avoid having scores that decrease as the number of documents, and thereby the dimensionality, we normalize the score according to the number of documents/dimensions in the index. In out implementation we use the function:

$$distanceScore(d) = e^{-ad}$$

This function will be 1.0 when the distance between the term and the cluster center is 0, while it will be 0.0 if the distance is infinite. The 'a' coefficient is a parameter that we will change how the correlation between each term is emphasized. We will vary this parameter in our test case to see how the performance is affected.

This ranking approach values terms that are close to the cluster center of the previous terms higher than those far away. We should therefore be able to suggest a set of related terms in all directions from the previous query terms, giving the user multiple options on how to expand the query.

Because a suggestion S will get discounted of the semantic correlation rank, we need to make an extension to the suggestion retrieval algorithm so that we are still guaranteed to retrieve suggestion in order of their final rank. Figure 37 outlines this extension.

```
persisitent:    suggestions = new PriorityQueue()
                termCorrelation
                suggestionSource
init( termCorrelation, suggesionSource )
        termCorrelation = termCorrelation
        suggestionSource = suggestionSource
getNextRank()
        while suggestionSource.getNextRank() >= getNextFinishedSuggestionRank()
                unfinishedSuggestion = suggestionSource.getNextSuggestion()
                corrRank = termCorrelation.computeCorrelationRank( unfinishedSuggestion )
                finishedSuggestion = updateSuggestion ( unfinishedSuggestion, corrRank )
                suggestions.add( finishedSuggestion )
        return getNextFinishedSuggestionRank()
getNextFinishedSuggestionRank()
        if suggestions.size > 0
                return suggestion.peek().getRank()
        return -2

getNextSugestion()
        return suggestions.pop()
```

Figure 37: An extension to suggestion retrieval to allow semantic term correlation discount.

5.2.3 Estimation of semantic term correlation by exploiting clusters.

Instead of using the previous matched terms cluster center as the basis for calculating the correlation with the last term, we will instead the previously matched terms distance to different clusters, as discussed in 3.4. Each term we match will carry a vector of discounts, between 0 and 1, for each cluster in the index. When more than one term is matched, we will simply multiply the relevance values for each cluster. In this way, we will calculate the semantic correlation discount for all the terms in a particular cluster before we even start looking for terms.

Figure 38 outlines a partial extension of the algorithm given in Figure 34. The function getNextTermNode() will be called when a termination node has been detected. With this extension active node carries a vector of the discount for each cluster that has been accumulated for previously

```
persistent:     clusterDiscountVector

init( clusterDiscountVector )
        clusterDicountVecot = clusterDiscountVector

getNextTermNode(  nodeQueue, leafTrieNode )
        nextClusterDiscountVector = pairwiseMultiply(
                                        clusterDiscountVector,
                                        leafTrieNode.clusterDiscountVector )
        nodeQueue.add( new ClusteredActiveNode(
                                queryContext.IndexRoot, nextClusterDiscountVector )   )
```

Figure 38: Partial algorithm used to calculate the next cluster discount vector.

matched terms. To move on to the next term, a new cluster vector used is calculated from the current cluster discount and the cluster discount of the term we have found. This is done by the pairwiseMultiply() function. The node produced here is a special active node that can produce the root node for each of the clusters. It will pass on the new clustering vector so that each of the descendant active nodes can be properly discounted based on which cluster it belongs to.

5.3 MATCHING EACH TERM SEPARATELY

One large concern with interactive fuzzy search is the running time of each character iteration. As the number of possible nodes increases exponentially with the query string, it would be very advantageous to be able to match each term separately instead of matching them as a large string in an infinite recursive tree index.

5.3.1 Tradeoffs for matching terms separately

To make matching each term separately possible we have to treat the space character as a special character that, instead of being matched in the regular process, is used to end a term and start a new one. We can then treat each term by a completely separate single term traversal of the index and separately retrieve suggestions. We will refer to a complete set of n suggestions that satisfy the n – 1 complete query terms and the last partial query term as a suggestion set. Our task will then be to find the k highest ranked suggestion sets.

One advantage of this arrangement over matching the multi term query as a long string is that each suggestion term will be ranked completely separately while the query terms are being matched, thus avoiding the ranking problem discussed above, where we had some serious restrictions on how terms could be ranked. Its first when the terms are combined into a multi term suggestion that we need to combine the rank and even then, we can choose from a larger variety of rank functions.

Another advantage is that it can be used to allow the edit quota per query term instead of for the entire query. It would in fact be difficult to implement in with a global edit distance quota, because we have no way of knowing which edit operations that leads to good suggestions because they are matched separately. Since having a large quota of edit distance is quite expensive because the number of nodes produced grows exponentially, this would increase the number of errors we are able to correct considerable. Moreover, it would arguable be more appropriate to match in this way, since a large edit quota could try to match terms with a very large edit distance to the actual query. If the edit distance for instance is 3 or more, it would be highly unlikely that it is this term the user intended to write unless that term is very long. It is conceivable to implement the same functionality when matching the multi term query as a long string by decreasing the number of edits performed in active nodes that start a new term. Some extra bookkeeping would however be needed, since we would have to keep track of the rank discount that we had occurred until that point.

The major disadvantage with this method is that it requires the assumption that the space character is correctly inserted, and we will therefore not be able to fuzzy match it. If we choose to either delete a space, substitute a space of insert a space, we will basically spawn a new query with a different number of terms. If we do this for every possible position in the query string, all the speed advantage this method has over the matching over a long string would probably be lost. In short, we have to know how many suggestions we are to retrieve for the complete suggestion set, and we have to know which part of the query string to be used to fill what position in the suggestion set.

5.3.2 Algorithmic description

Since we are doing n single term index traversals, where n is the number of query terms, we can simply re-use our previously developed single term algorithms to find index terms. We will also add a suggestion caching facility around our previous approach, so that the (n – 1) complete terms can reuse their previous work for each character iteration. This is a valid approach, since they will not be modified, and they will therefore produce the same sequence of index terms each time. Pseudo-code for the caching facility is sketched out in Figure 39. We also include a new method getSuggestion(index) which get the suggestion in the ordered sequence at position index. It retrieves it from the cache if it is there. If it is not it explores units of the index until it is found, or there is no more parts of the index to explore.

When producing suggestions sets, it is important to distinguish between the rank estimate and the exact rank of the set. The rank estimate is some rank calculated from the individual suggestion terms rank. Unlike the situation in the previous section, we can use a plethora of ways to calculate this rank, as we are not limited by a rule that states that the combined rank must be non-decreasing. It only need to be so that if the average term rank goes up, the entire rank estimate goes up.

The exact rank is the rank we have after we have calculated the semantic correlation between the suggestion terms. This exact rank must be smaller than the estimate rank. The estimate rank is thus an optimistic estimate of the rank. While this is true, we can produce the suggestions sets with the best estimates first, which, with no further knowledge present, is presumed to have a larger chance to have the higher rank. We then calculate the exact rank, and put the set in a priority queue of finished suggestion sets. If one of the sets in a queue has a higher rank than the next suggestion set we can produce, then it can popped from the queue and returned as a suggestion to the user. If not, we need to produce the next suggestion set.

```
persistent:     query
                suggestionCache = { }
                nextSuggestion = 0
init( query ) query = query

updateQueryString( nextCharacter )
    nextSuggestion = 0
    if nextCharacter != null
        suggestionCache.clear()
        query.updateQueryString( nextCharacter )

peekNextNodeRank()
    return query.peekNextNodeRank()
exploreNextNode()
    return query.exploreNextNode()
peekNextSuggestionRank()
    if nextSuggestion < suggestionCache.size()
        return suggestionCache[nextSuggestion].getRank()
    return query.peekNextSuggestionRank()

getNextAvailableSuggestion()
    suggestion = null
    if nextSuggestion < suggestionCache.size()
        suggestion = suggestionCache[nextSuggestion]

    else
        suggestion = query.getNextAvailableSuggestion()
        suggestionCache.add( suggestion )
    nextSuggestion += 1
    return suggestion
getSuggestion( int index )
    while peekNextSuggestionRank() >= peekNextNodeRank() AND
             suggestionCache.size() <= index
        getNextAvailableSuggestion()
    while suggestionCache.size() <= index AND peekNextNodeRank() != -1
        exploreNextNode()
        while peekNextSuggestionRank() >= peekNextNodeRank() AND
                suggestionCache.size() <= index
                getNextAvailableSuggestion()
        if suggestionCache.size() <= index
            return null
        return suggestionCache.get( index )
```

Figure 39: Pseudo-code for a caching facility.

The first suggestion set produced will be the set of the highest ranked suggestions from each of the query terms. This node will be put into a priority queue sorted by rank estimate. The algorithm will then iterate on this queue by popping the estimated best suggestion set. It will calculate the exact rank and put it into the queue of finished suggestion sets. It will also produce the children suggestion sets of the current suggestion set and put them into the queue of unfinished suggestion sets.

A child of a suggestion set is a suggestion set that modifies its parent set by replacing one of the sets suggestions with the next suggestion for the query of that query term. To avoid duplicating suggestion sets, a parent will be limited to modifying the suggestions to the right of and including the term that was modified to produce that parent. An example of which children can be produced by which parent is given in Figure 41. It is based upon a multi term query of two terms: A B. These query terms have 3 suggestions each, as illustrated in Figure 40.

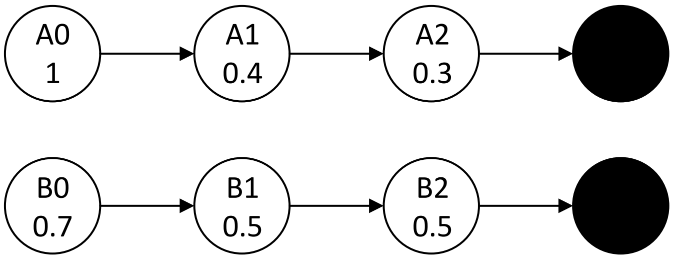


Figure 40: The suggestions for terms A and B and their respective ranks.

We see that the left child of the root in Figure 41 produces all the suggestions from A1 and B0 where both suggestions in the set are replaced with a later suggestion, while the right child only produces suggestions after A0 B1 where the suggestion for B is replaced. This avoids duplicates as we would for instance have gotten two equal versions of A1 B1 if both the left and the right child of the root were allowed to modify the A suggestion. Pseudo-code for this algorithm is given in Figure 42, while the produceChildren() function of a suggestion set is given in Figure 43.

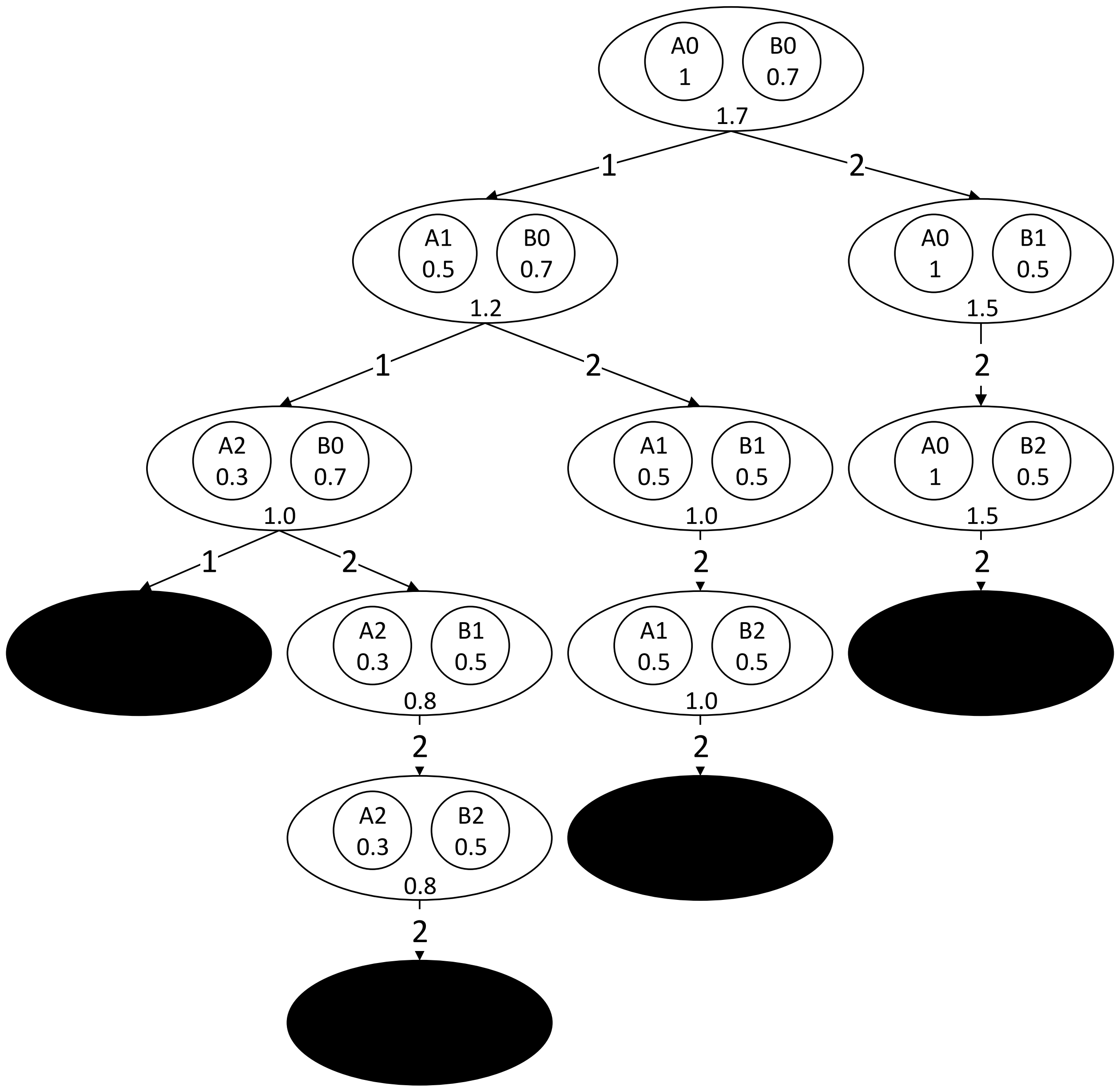


Figure 41: The skewed tree structure of suggestion set production.

```
persistent:     termSeparator = ' '
                finishedSuggestionSets = new PriorityQueue()
                unfinishedSuggestionSets = new PriorityQueue()
                singleTermQueries = { }
init()
        addNewQueryTerm()

updateQueryString( nextCharacter )
        if nextCharacter == termSeparator
                addNewQueryTerm()
        else
                singleTermQueries.last.updateQueryString( nextCharacter )
        finishedSuggestionsSets.clear()
        unfinishedSuggestionSets.clear()
        produceFirstSuggestionSet( unfinishedSuggestionSets, singleTermQueries )

addNewQueryTerm()
        singleTermQueries.add( new CachedQuery( new PriorityQuery() )

peekNextNodeRank()
        if unfinishedSuggestionSets.size > 0
                return unfinishedSuggestionSets.peek().getRankEstimate()
        return -1

exploreNextNode()
        currentSuggestion = unfinishedSuggestionsSets.pop()
        currentSuggestion.calculateExactRank()
        finishedSuggestionSets.add ( currentSuggestion )
        currentSuggestion.produceChildren( unfinishedSuggestionSets, singleTermQueries)

peekNextSuggestionRank()
        if finishedSuggesitonSets.size() > 0
                return finishedSuggestionSets.peek().getRank()
        return -2

getNextAvailableSuggestion()
        return finishedSuggestionSets.pop().produceSuggestion()
```

Figure 42: Pseudo-code for separate term matching.

```
persistent:     suggestionSet
                suggestionIndecies
                numberOfSuggestions
                productionStart

init( suggestionSet, suggestionIndecies, numberOfSuggestions, productionStart )
    suggestionSet = suggestionSet
    suggestionIndecies = suggestionIndecies
    numberOfSuggestions = numberOfSuggestions
    productionStart = productionStart

produceChildren( queue, singleTermQueries )
    for i = productionStart to numberOfSuggestions – 1
        newSuggestionIndecies = { }
        foreach index in suggestionIndecies
            newSuggestionIndecies.add( index )
        newSuggesitonIndecies[i] += 1
        newSuggestion = singleTermQueries[i].getSuggestion( newSuggestionIndecies[i] )
        if newSuggestion != null
            newSuggestionSet = suggestionSet.copy()
            newSuggestionSet[i] = newSuggestion
            queue.add( new SuggestionSet( newSuggestionSet, newSuggestionIndecies
                                          numberOfSuggestions, i ))
```
Figure 43:Pseudo-code for SuggestionSet.

We see that the updateQueryString() function clears all the queues and produces the first suggestion set of the new character iteration. It updates the current partial query term. The extract next node function will process one unfinished suggestion set. It will calculate its true rank and extract all its children. Note that we will not be able to extract the best child suggestion set and produce the next set later, because we have to produce the nodes before we know the exact rank. We therefore cannot actually know which child is best before all are produced. The function used to calculate the true rank is the one used for calculating direct semantic term correlation which we presented in 5.2.2.

6 TEST DESCRIPTION

In this section we outline how we will test the performance of the algorithms we have presented in the previous chapters. As we stated in the introduction, this thesis focuses on the run time performance of the algorithms and we will therefore not perform any tests relating to the quality of the suggestions that the algorithms makes. The algorithms are however, as we have described, based on similar ranking functions and should therefore retrieve very similar results.

6.1 IMPLEMENTATION DETAILS

The algorithms are implemented in java as a loosely coupled extension to Apache Lucene 3.6.0[3]. It works by making a clustered index structure as presented in chapter 0 from the terms indexed in a specified Lucene index. The index structure support being read and written to disk, but the entire index needs to be in index at query time. This requirement is there because our implementation of the index structure is heavily object oriented, and java is relatively slow when deserializing these objects.

The clustering used to cluster terms are a custom written adaptation of the k-means algorithm. The terms are clustered by using a vector space model of the documents, where each term is represented by the tf-idf weights of the related documents (calculated by Lucene). The code for this is not optimized to be especially fast, so we will not evaluate the indexing speed.

The inverted index stored in the terminating nodes of the index will be used to calculate the term correlation discount for the direct long string and separate terms algorithms. We will make a single cluster indexes to avoid the overhead of using clusters for these algorithms. Note that all the algorithms will use the same implementation for trie nodes. The leaf nodes will thus carry an inverted index even when it’s not needed. The cluster relevance vector will also be there, but in the absence of multiple clusters, it will only have one element, and its effect will therefore probably be negligible (a 64 bit reference pointing to a 32 bit integer for each leaf node)). The leaf nodes will also redundantly store the term it represents. This is done make the implementation of suggestion retrieval easier and more efficient at query time.

6.2 TEST DATA

The document collection used for testing is Medline 2004, which is a collection of documents related to life sciences. We are indexing the abstract and the title field of these documents, which are the textual fields (the other fields describes document origin, publishing time etc.). The data of the single cluster term index is given in Table 6.

Dataset	Medline 2004
Number of documents	4591008
Size	5.081 GB
Index size, single cluster	430 MB
Indexed terms	1530488

Table 6: The data on the trie index that consists of Medline 2004 documents.

[3] Apache Lucene are a trademark of the Apache Software Foundation (see http://lucene.apache.org/).

Test queries will be extracted at random from the indexed terms. Some amount of errors will then be introduced into these terms randomly, where the random inserts of substitutions will get a random character between ‘a’ and ‘z’. For multi term test queries, multiple random terms are simply concatenated, separated by the space character.

Preliminary tests has shown that the simple algorithm is very slow on the full index, and we will perform a limited numbers of queries when testing it to make the tests tractable. The number of queries we are able to perform will also be limited when increasing the allowed edit distance because the time usage for the simple algorithm will grow very large. We will therefore test with a much smaller index when comparing all three algorithms, and rerun the tests on the full index for the priority algorithms. The direct long string algorithm were also very slow and we had to make similar simplification for it in the multi term test cases.

6.3 Measurements and source of error.

The time usage of the queries will be measured using the System.nanoTime() method that the java[4] language provides. It gives the current wall time in nanoseconds. The result will be written to disk between queries. This might still have an effect on the queries, as it might invalidate the CPU cache. The result files (one for each algorithm tested in a test case) will be kept open during the tests to avoid expensive I/O operations taking up a significant amount of time, so they will contribute to a slight system resource overhead.

There are multiple sources of measurement errors present. First, java is a garbage collected language, and the garbage collector may run at any time, and thereby affecting the results. Since java is a JIT compiled language, we will also see higher run times the first time the different functions are called. We will therefore perform 5 dummy queries before each test, to avoid polluting the result with start-up performance hurdles. We also pre-allocate a large heap size to avoid the large cost of expanding the heap. Moreover, the test machine may have background processes that may also consume resources.

Another source of error is the nature of picking random index terms as query data. This should not be a problem if the number of queries is large enough, but it may have a large effect in those cases where we have lowered the number of queries to make the test tractable. These cases do however generally use so much time compared to the other algorithms that the exact results are of no real interest.

The implementation of prefix based query contains a known bug in a corner case around terminated nodes. This will cause it to sometimes have one suggestion that the query string based algorithm does not have and visa versa. This will only show up in a small portion of the queries, at least as far as we have been able to verify, and should not have a significant impact on the result.

6.4 Test machine

The test will run on a machine with the following relevant specs:

- Intel Core i5-2500K
- 24 GB DDR 3 ram
- Intel SSD 320 120GB (the documents is fetched from this disk)

[4] Java is a trademark of Oracle (see http://www.oracle.com/technetwork/java/index.html).

- Samsung HD103SJ 1000 GB (the results are written here)
- Windows 8
- Java runtime 1.7.0.09

6.5 TEST CASES

Here we will present the different test cases we will use and outline what we are trying to measure. These cases will present the index in use, the number of queries and the values we will use for the variable parameters that we will test against. If nothing else is specified the tests will be performed with maximum edit distance of 1 and with the requirement of retrieving the 20 best suggestions. Every test will be performed by actually retrieving 20 suggestions for each character iteration, but all test except the incremental suggestion test will present the average time use for the entire query.

6.5.1 Edit distance scaling test

Index	50000 terms: all single term algorithms All terms: priority algorithms only

Number of queries	Priority algorithms: {10000, 1000, 100}
	Simple algorithm: {100, 50, 10}

Allowed edit distance	1, 2, 3

Table 7: Test specification for the edit distance scaling test.

The purpose of this test is to measure how well the algorithms scale with allowed edit distance. It will test max edit distance 1, 2 and 3. Because of the time increasing time usage, we decrease the numbers of queries performed for max edit distance 2 and 3 (1000 and 100 for the priority algorithms and 50 and 10 for the simple algorithm). Moreover, the preliminary tests showed that the simple algorithm will use extreme amounts of memory for max edit distance 3, so we will only use 50000 terms when comparing all three algorithms. We also include a test of the priority algorithms for the full index to test the feasibility of higher allowed edit distances for large corpuses.

6.5.2 Incremental suggestions test

Index	All terms
Number of queries	Priority: 10000 Simple: 50
Term lengths	1 to 6

Table 8: Test specification for the incremental suggestions test.

In this test the individual times of each character iteration is measured. This is to see where the algorithms uses the most time. Because we want the user to perceive that the suggestions are given instantly, there are a limit of about 100ms of time for each character iteration [8]. It is therefore desirable to avoid time usage spikes for some of the character iterations while others are fast. Again, we need to limit the simple algorithms to 50 terms, because the time usage of these queries on the full index is very large.

6.5.3 Index size scaling test

Index	100000, 200000, 300000, 400000, 500000 terms
Number of queries	Priority: 10000 Simple: 50

Table 9: Test specification for the index scaling test.

This test is to see how the algorithm scales with increasing index sizes. The simple algorithm will intuitively do quite badly as the index size increase due to it exploring every path in the tree that are within the allowed edit distance. The hypothesis is that the priority algorithms will scale much better due to being able to prune away large portions of the index.

6.5.4 Number of suggestions to retrieve test

Index	All terms
Number of queries	Priority: 10000 Simple: 50
Suggestions to retrieve	1, 20, 40, 60, 80, 100

Table 10: Test specification for the number of suggestions to retrieve test.

This test measures how the algorithms scale with the number of required suggestions. Because the simple algorithm retrieves every suggestion, we don't expect it to use any more time to retrieve more suggestions. The priority algorithms are however inherently lazy when retrieving suggestion, and this may result in increasing time usage as the number of suggestion to retrieve increases.

6.5.5 Number of clusters scaling test

Index	All terms.
Number of queries	Priority: 10000
Number of Clusters	1, 20, 40, 60, 80, 100

Table 11: Test specification for the number of clusters scaling test.

This test measures the impact clustering, as presented in section 3.4, makes on a regular single term query. We will also include the sizes of the different indexes to see how the index size scales with the increased number of clusters. We will only use the priority algorithms in this test because we really only wish to see if a clustered trie causes a significant slowdown in the queries and not measure the differences between the algorithms.

6.5.6 Multi term scaling test

Index	All terms
Number of queries	Separate and Clustered: 1000 Direct: 1000, 1000, 100, 25
Number of terms	1, 2, 3, 4

Table 12: Test specification for the multi term scaling test.

This test will compare the performance of the three different techniques we have presented for extending the single term algorithms to handle multi term queries. The query string based priority algorithm will be the basis for the algorithms in this test. We will test with increasing numbers of terms to see how the algorithms handles increasing query length.

We had to limit the number of queries done for the direct long string algorithm, because preliminary tests reviled that they used a very long time to complete the cases with 3 and 4 terms.

The discount parameter for semantic correlation used in this test is 1, which in most cases will result in semantic correlation discounts between 0.9 and 1.0.

6.5.7 Multi term discount test.

Index	All terms. Clustered with 60 clusters and single cluster variants.
Number of queries	10000
Number of terms	2
Discount parameters	0, 20, 40, 60, 80, 100

Table 13: Test specification for the multi term discount test.

This test will compare the different multi term algorithms when the using larger discounts for the semantic correlation between terms. The cluster based algorithm will use an index with 60 clusters. Because the clustering discount is hardcoded into the index, 6 different clustered indices with the specified discount parameters will be used. The direct long string and separate terms algorithm will use a single clustered index. We expect to see that the two algorithms calculating the term correlation after the suggestions are found will have a much harder time because the actual rank of a suggestion can be much lower than the estimated rank. The result of this test is interesting, because it will outline some restrictions on how a ranking function can behave in order to avoid sacrificing performance.

The discount parameter 0 causes all semantic correlation discounts to be 1.0, which is the same as having no discount. With the value 100, we saw discount values down to about 0.2.

7 RESULTS

In this chapter we will present the results from the tests we described in 6.5 and discuss their implications. The results will be presented in graphs and the time unit used are micro seconds.

7.1 EDIT DISTANCE SCALING

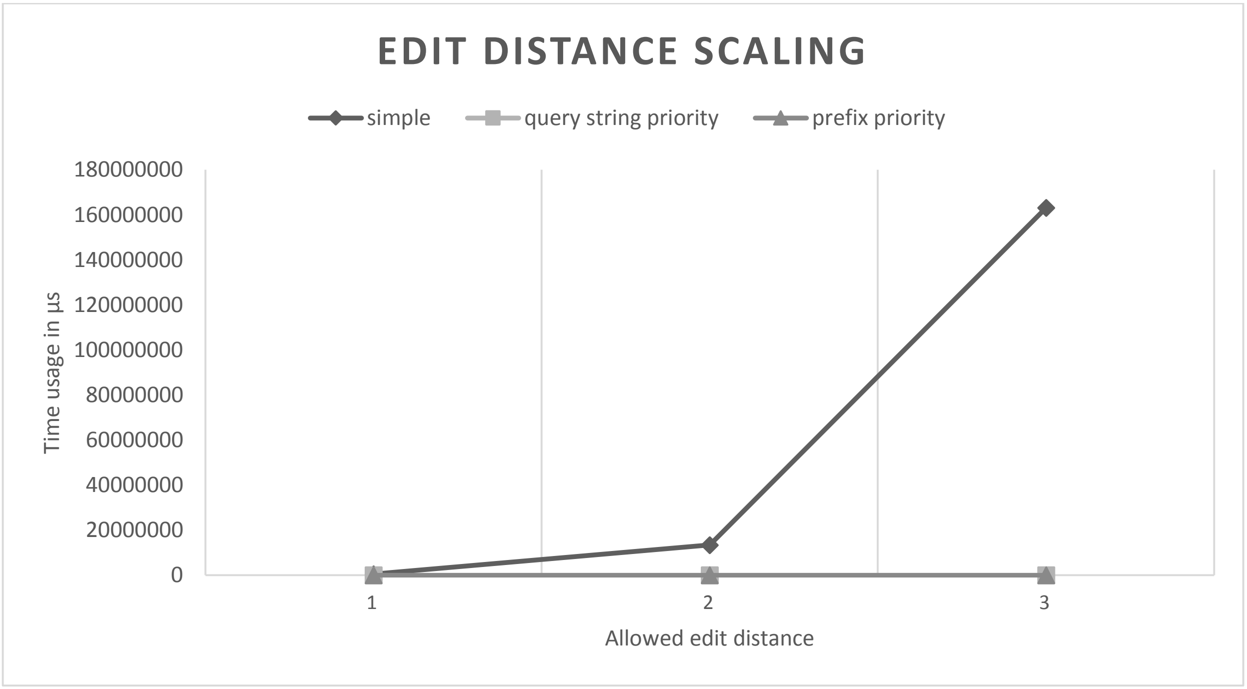


Figure 44: Results for the edit distance scaling test.

The results of the edit distance scaling test is given in Figure 44. What we can instantly see is that the simple algorithm scales very poorly with the increases in edit distance. At edit distance 2 it is using over 13 seconds to complete the query (which clearly will be unable to give the user an instant response). At edit distance 3 it gets even worse with a time usage of 180 seconds on average. The simple algorithm is clearly not suited for large edit distances even on this relatively small index of 50000 terms. The priority algorithms uses so little time compared to the simple algorithm that they are hardly visible. We therefore removed the simple algorithm results in Figure 45. Both perform very well even for edit distance 3, even if the increase in time usage is quite large compared to edit distance 1 and 2. We see this trend continue for the full index in Figure 46. Here, the prefix priority algorithm does however use 700ms on average for edit distance 3, which is on the threshold of true instant response for the individual characters. The priory algorithm here performs notably better hovering just above 100ms.

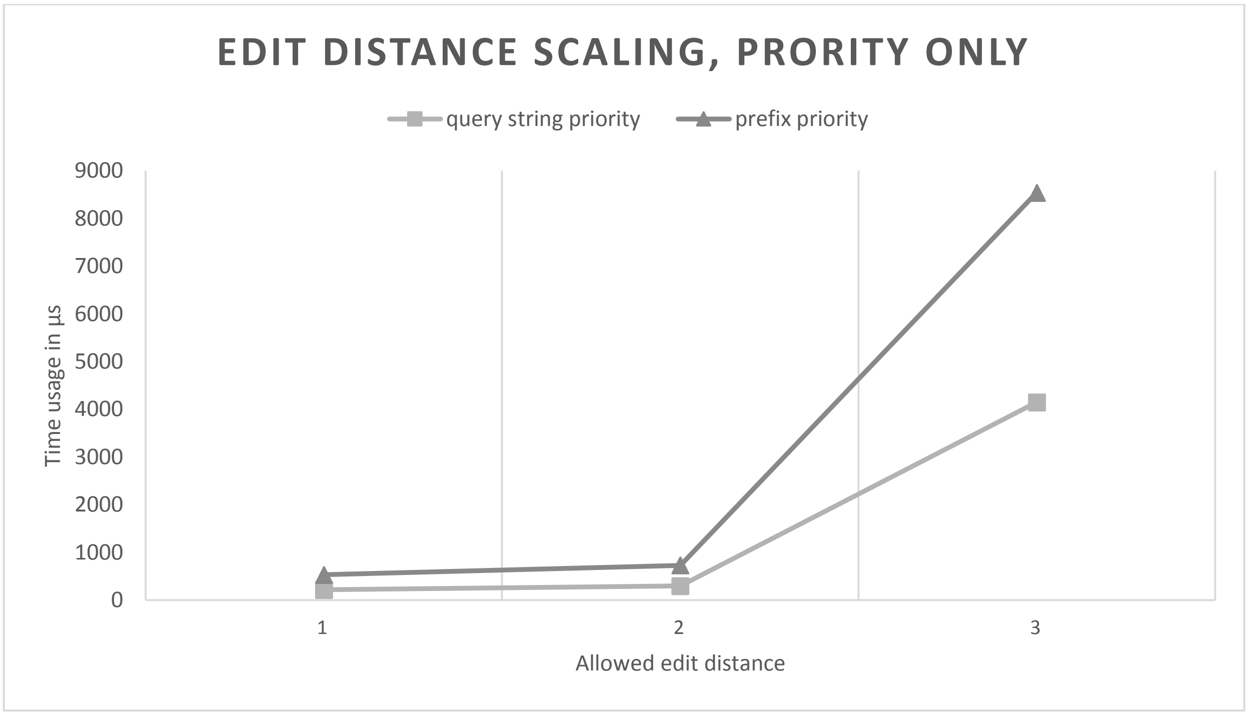


Figure 45: Results for the edit distance test, priority only.

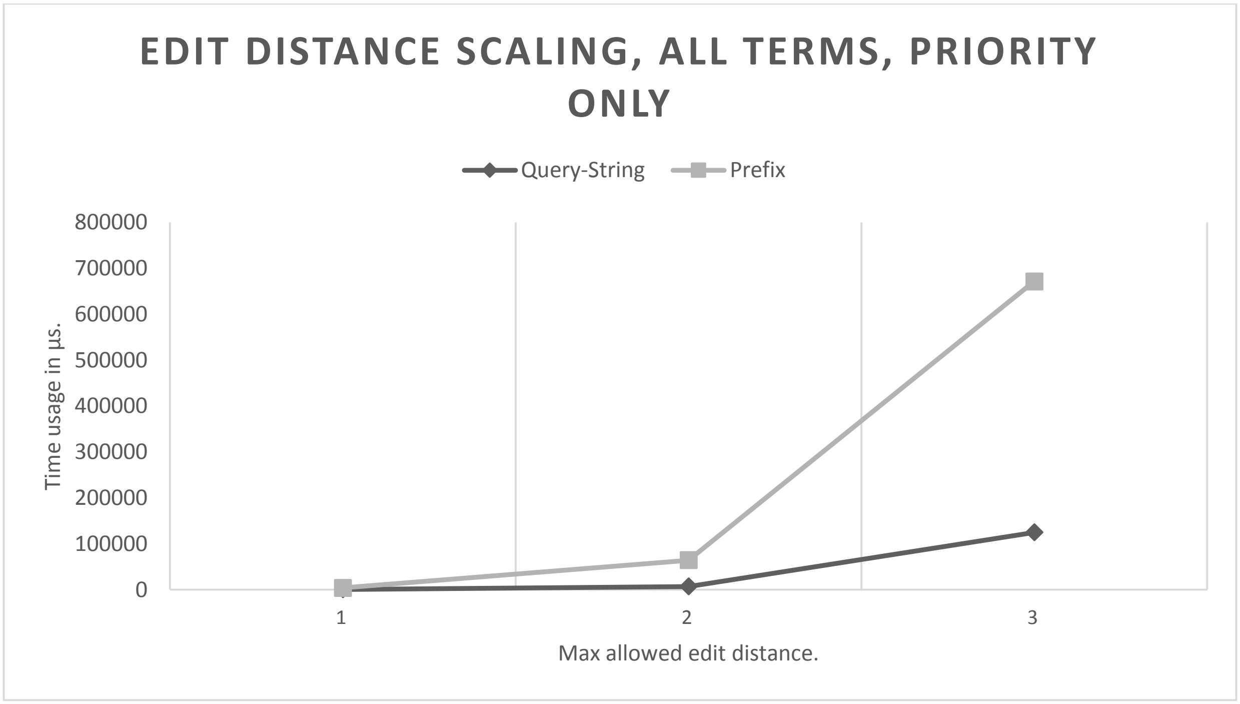


Figure 46: Results for the edit distance test on the full index, priority only,

7.2 ITERATIVE SUGGESTIONS

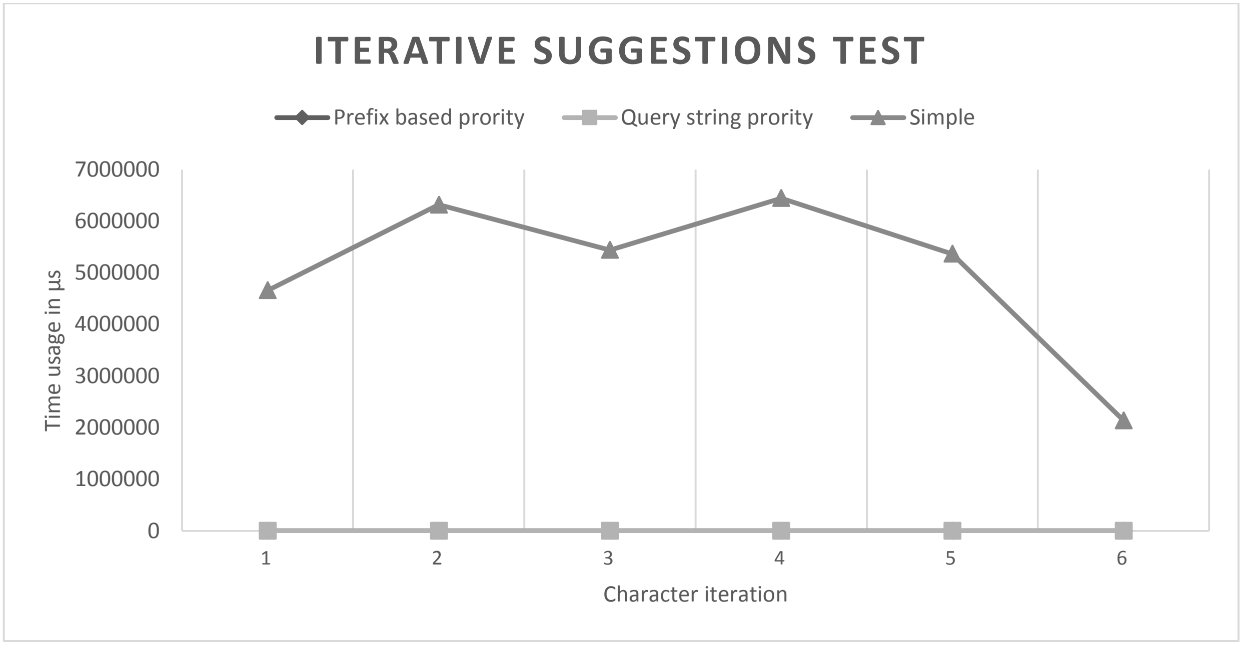


Figure 47: Results for the iterative suggestions test.

We see that the simple algorithm spends too much time on every iteration that neither could be considered responsive, as they are far above 100ms. Each iteration takes over 2 seconds with the worst cases using over 6 seconds. The interesting bit is that the time usage drops significantly for the last iteration. This may be because most of the index has been discarded for being further away from the query string than 1 in the previous iterations. Since the priority algorithms comparatively uses very little time, we have removed the results of the simple algorithm in Figure 48.

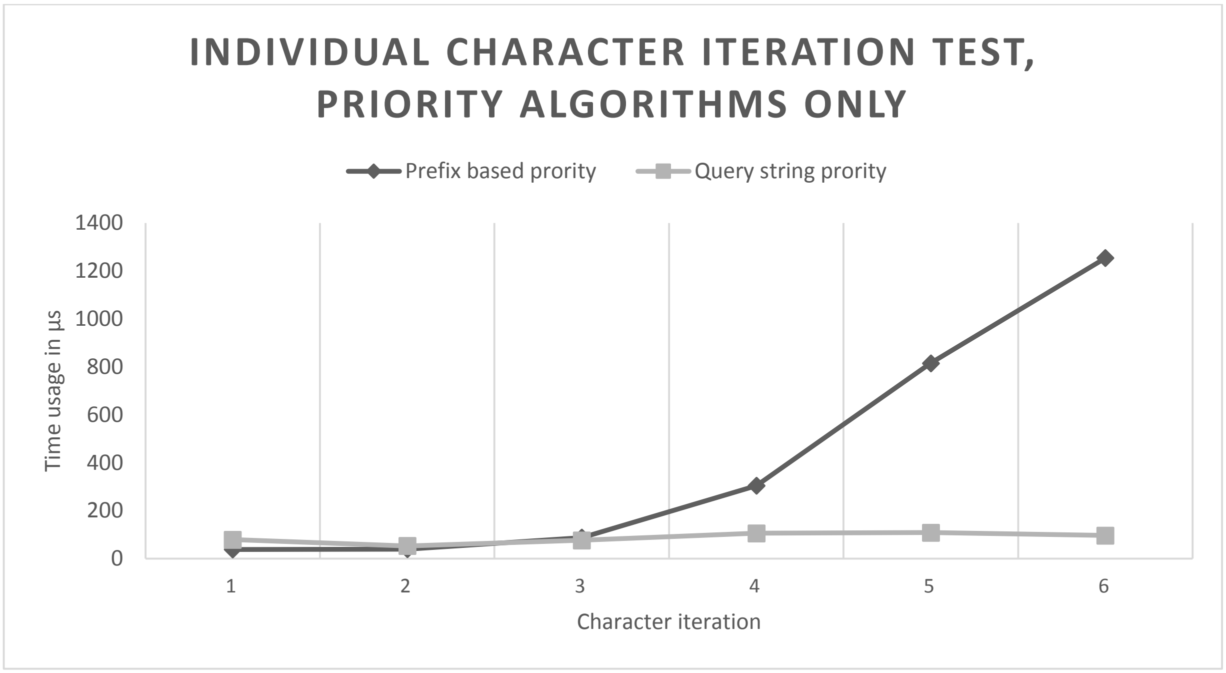


Figure 48: Results for the iterative suggestions test, priority algorithms only.

While both algorithms performs excellent in their absolute time usage, we see that the prefix based algorithm, while slightly beating the query string based algorithm for the first 2 iterations, scales quite poorly for further iterations. The query string based algorithms does, however, have no clear trend and seems to be quite stable. The reason why the prefix based algorithm scales so poorly is probably due to the fact that it cannot predict the rank of the children of an active node before they are actually produced. It might therefore need to produce many nodes it doesn't need, whereas the query string based algorithm only needs to materialize active nodes that it actually needs to explore (with the possible exception of the first insert and the delete nodes). This means that even if the query string based algorithm needs to handle duplicates, its ability to prune away parts of the index might be more powerful.

7.3 Index size scaling

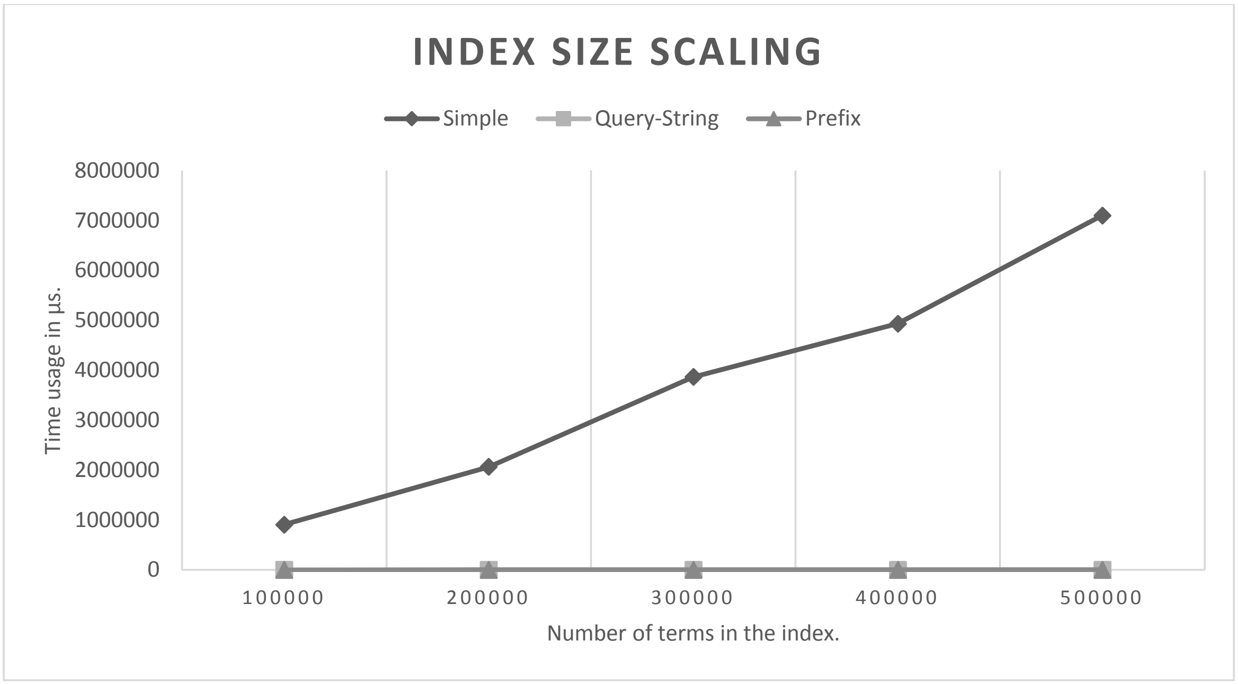


Figure 49: Results of the index size scaling test.

Figure 49 shows the time usage scaling of the algorithms when the index size is increase. We see the same trend as we did in the last two sections. The simple algorithm is simply outperformed by the priority algorithms. Moreover, we see that its time usage scales linearly to the number of terms in the index. Again we remove the simple algorithm results from the result set and arrive at the graph in Figure 50. Again we see that the query string based algorithm scales worse than the query string based algorithm. While the priority algorithm shows a very clear trend towards higher time usage when the number of terms increases, the query string based show only a very tiny increase. What we can take away from this is that the query string based algorithm scales very well with increased index size, and should be able to handle very large indexes in real time.

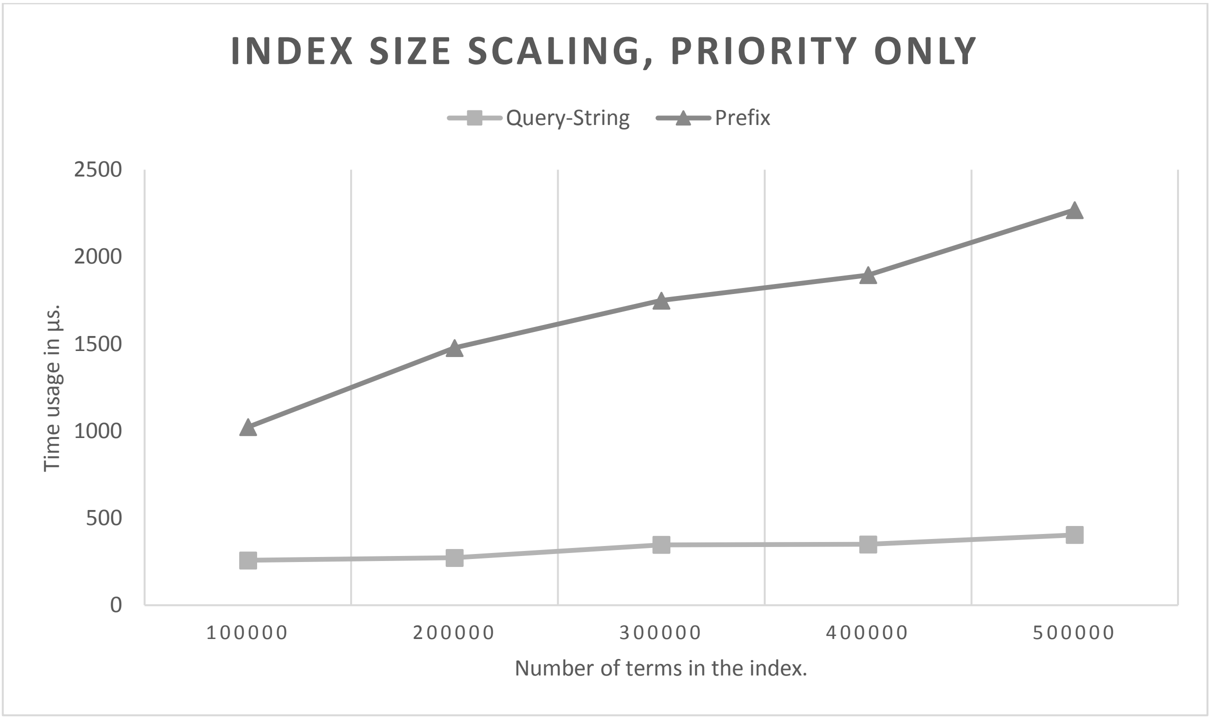


Figure 50: Result of the index size scaling test, priority only.

7.4 NUMBER OF SUGGESTIONS TO RETRIEVE

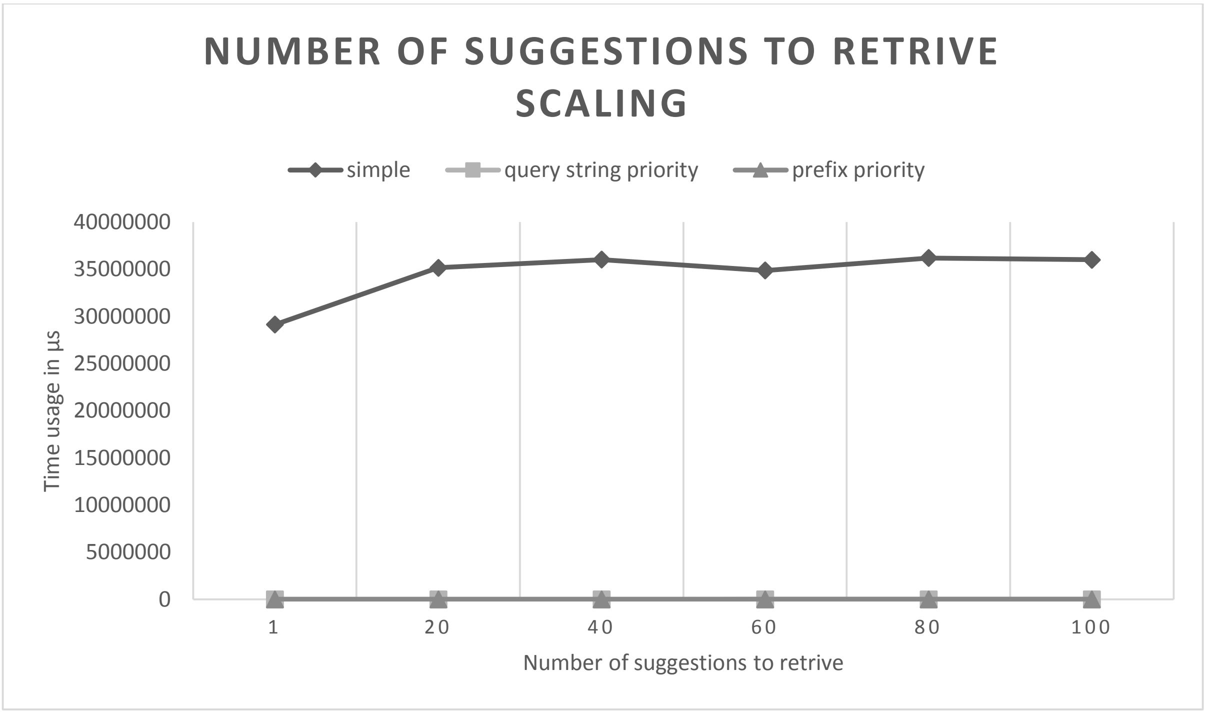


Figure 51: Result of the number of suggestions to retrieve scaling test.

Figure 51 shows the time usage of the algorithms while we increase the number of suggestions to retrieve. Figure 52 shows the same results after removing the results of the simple algorithm. As expected, we see no clear trend of an increase of the time usage for the simple algorithm. Since the already retrieves every suggestion, there are little extra work involved in returning more of them.

The query string based algorithm seems to scale linearly when increasing the number of suggestions, while the prefix based seems to have a high increase in the start, while the trend seemingly flattens as the number of suggestions increases father. In fact, we see that the difference between the two algorithms gets slightly less as the number of suggestions gets large. The reason for this trend is likely to be related to the reason why the string based algorithm is faster than the prefix based in the first place. The nodes that the prefix based algorithm needs to materialize to find the best child node of an active node might become useful once we need more suggestions.

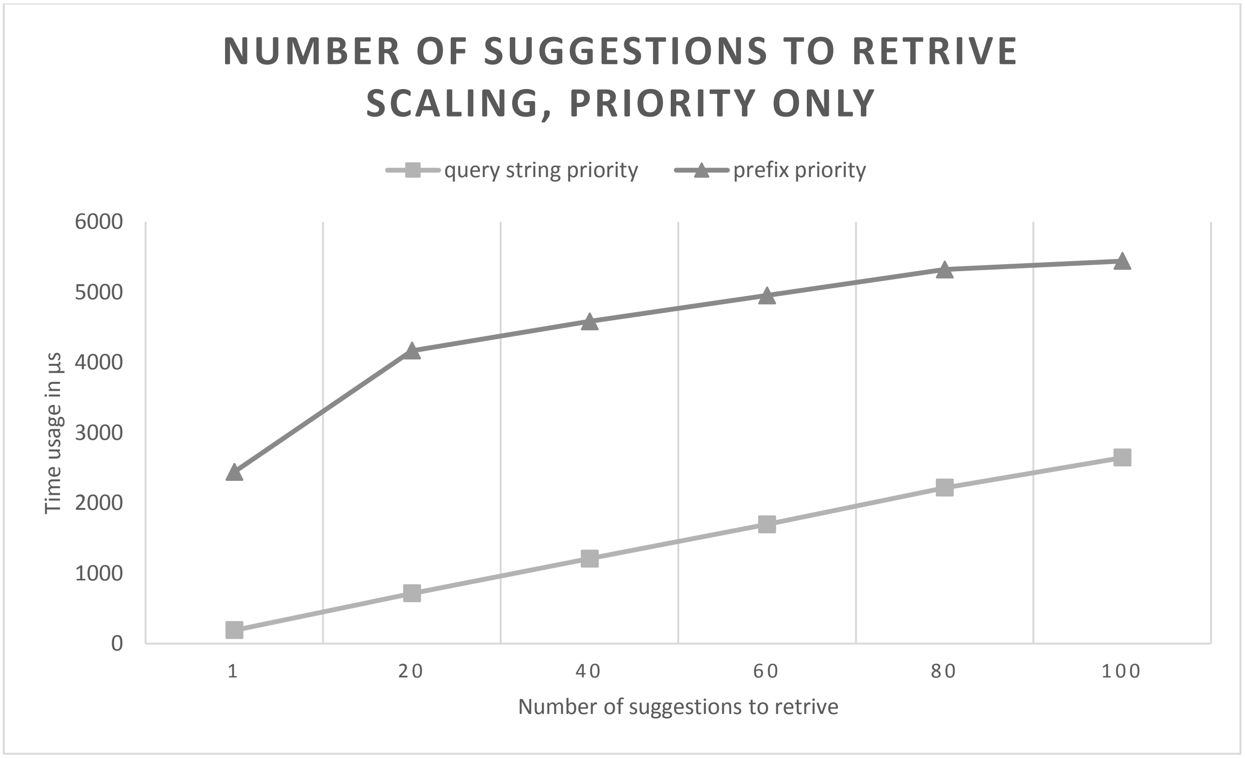


Figure 52: Result of the number of suggestions to retrieve scaling test, priority algorithm only.

To find out if the trend in Figure 52 would continue, we reran the test for the priority algorithms retrieving 100, 1000 and 10000 suggestions. The result of this run is given in Figure 53. We see that the prefix based algorithm indeed grows slower than the query string based algorithm, and at 10000 suggestions, the prefix algorithm will actually be significantly faster than the query string algorithm. This seems to suggest that once the prefix algorithm has a use of all the nodes it has to create in order to find the best child of an active node it will actually be quite efficient, most likely because of its lack of overlapping active nodes, as explained in 4.9, and thereby both avoid having to deal with duplicates.

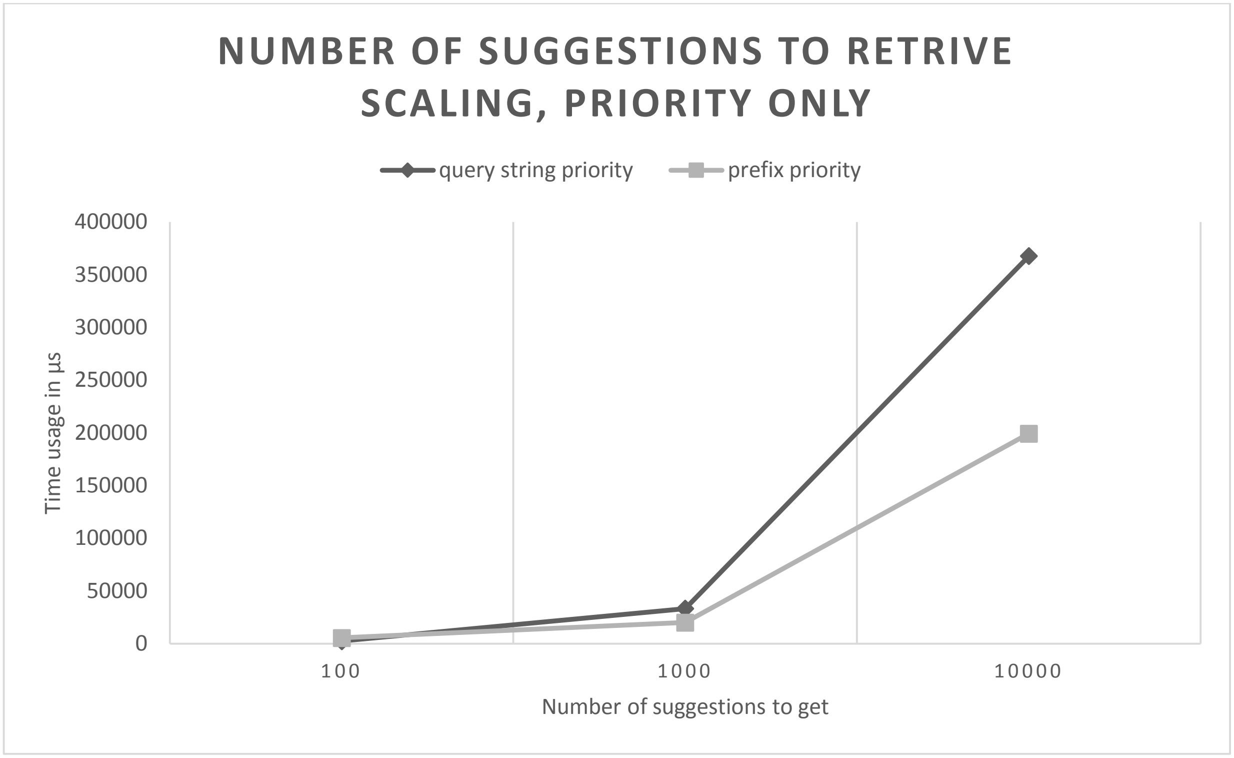


Figure 53: Rerun of the number of suggestion retrieval test with more suggestions to retrieve.

7.5 CLUSTER SCALING TEST

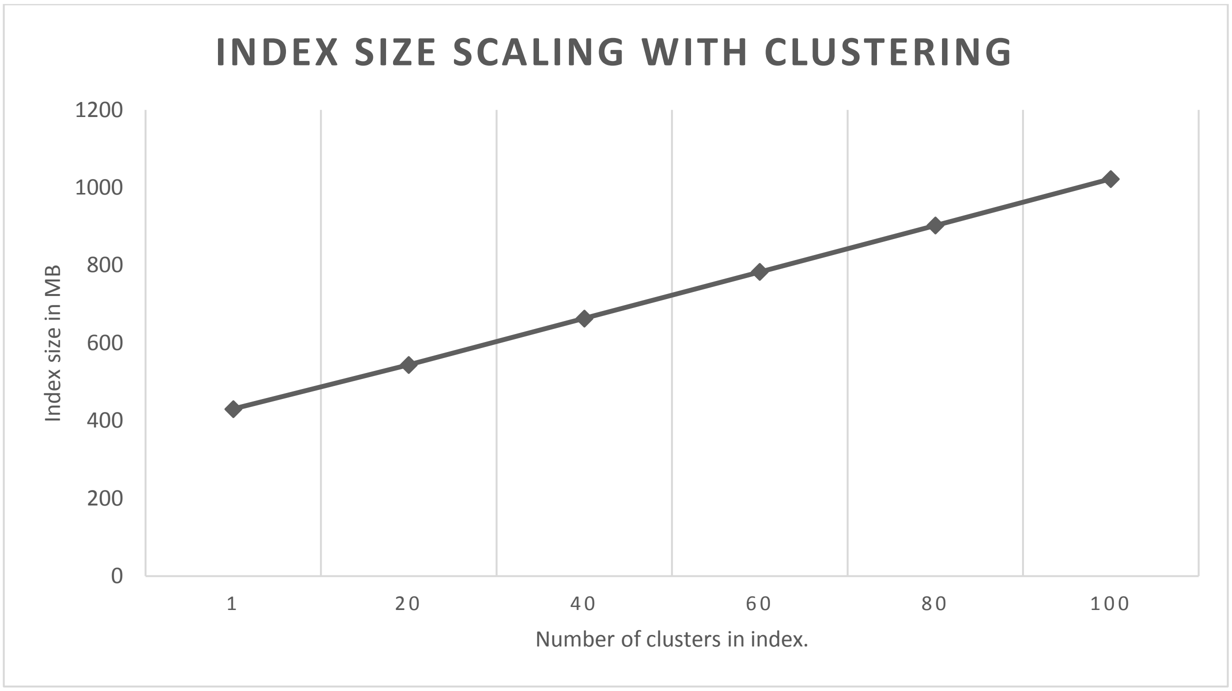


Figure 54: Index size scaling with clustering.

In this test, we looked at how the priority algorithms perform when we use an index of multiple clusters. Figure 54 shows how the size of the indexes themselves increase in size as the number of clusters increase. We can see a linear that increase over the base size as we increase the number of clusters. This increase is due to the loss of the inherent compression in the trie data structure that we get when terms are no longer able to share storage for common prefixes, as we mentioned in 3.4.

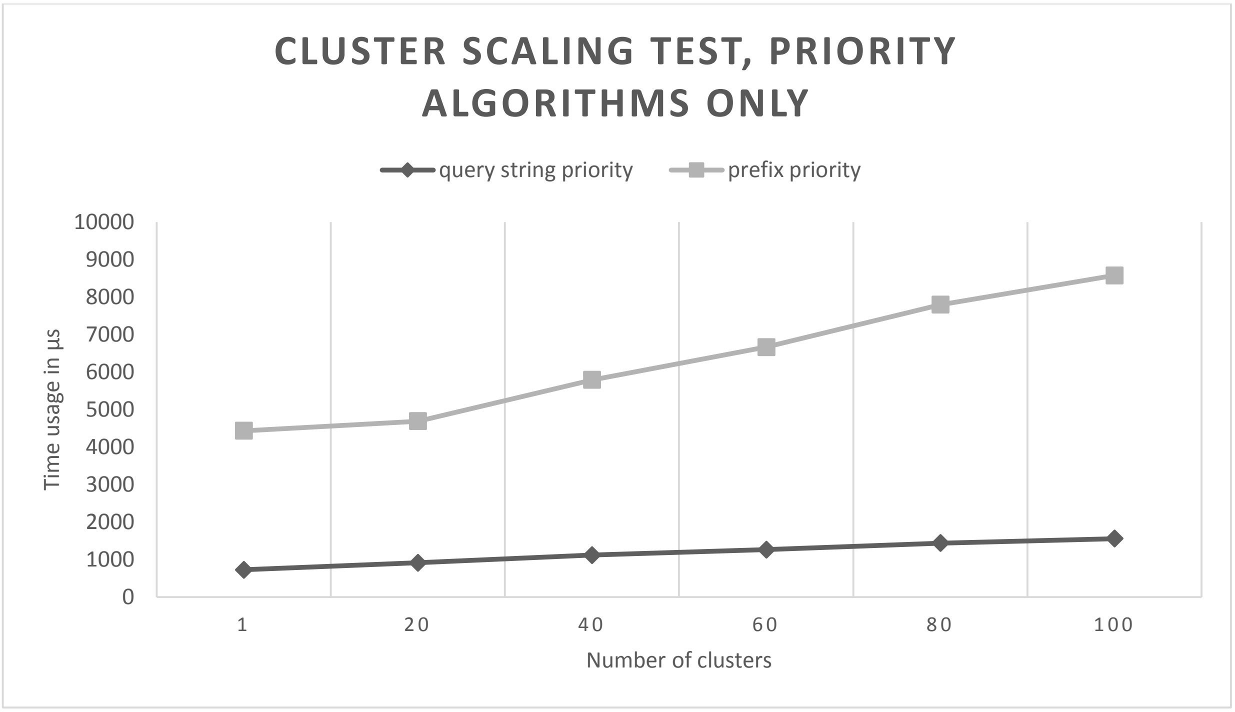


Figure 55: Results of the cluster scaling test.

As Figure 55 shows, the loss in compression also affects the queries. While both algorithms takes a performance hit, we again see that the prefix based algorithm struggles to scale as gracefully as the query string based algorithm because it needs to materialize more nodes.

7.6 MULTI TERM SCALING

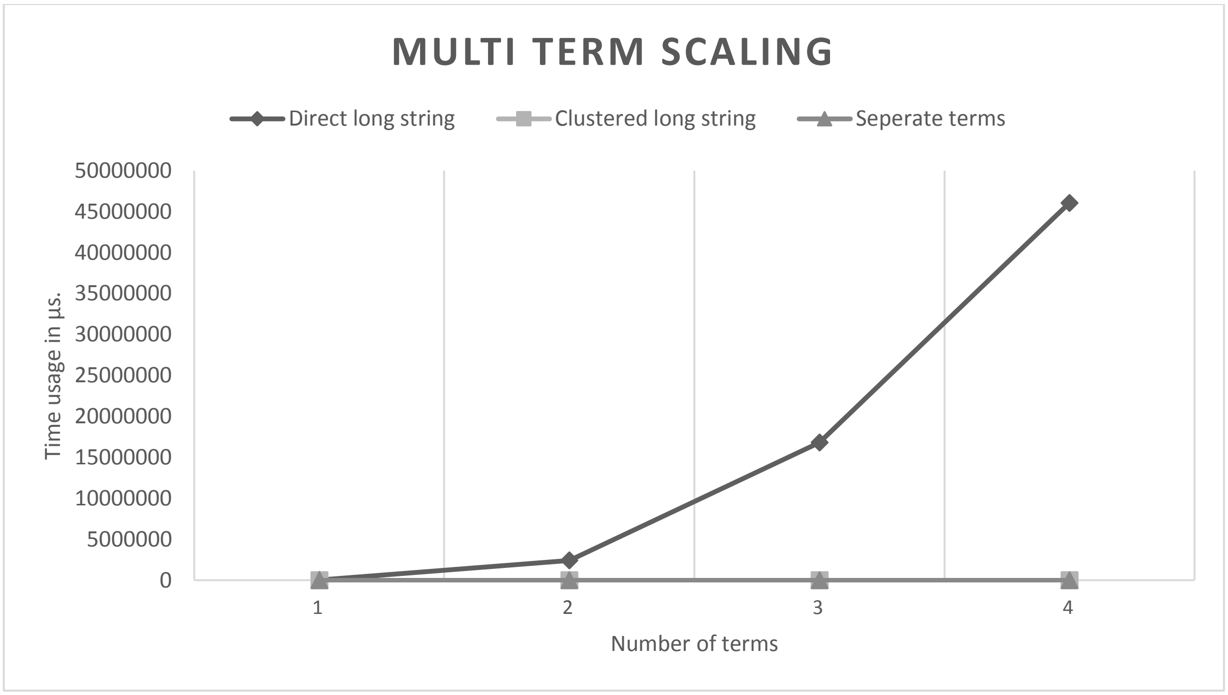


Figure 56: Results of the multi term scaling test.

Figure 56 show how the average time usage of the three multi term algorithms scales with the number of terms. It's immediately obvious that the direct long string algorithms scales very poorly and is basically useless for our purpose. Using between 10 and 20 seconds to retrieve 20 suggestions for a 3 term query, and almost 50 seconds for a 4 term query is very far from a real time interactive query. Even worse, the trend seems to be exponential. On the other hand, Figure 57 shows that both the clustered long string algorithm and the separate term algorithm does very well and performs nearly identically using under 10ms on average for the entire query even for 4 terms. The separate terms algorithm does seem to scale worse than the crusted algorithm, but both are very viable option for performing interactive fuzzy query expansion. The reason the separate terms algorithm fares so much better than the direct long string algorithm is likely to be the caching facility that we were able to device for it, making the generation of the different suggestion sets that needs to be evaluated to find the top-k a much easier task. It is also does not need to carry as much state as the long string algorithms needs to carry, as the last partial query will be treated without the baggage of the previous query terms.

While both of these algorithms have certain restrictions compared to the direct long string algorithm, as we discussed in chapter 5, they are able to answer multi term queries in real time, which we were not able to do with the direct long string algorithm. Such restrictions may therefore be necessary to make a viable algorithm for our purpose. It is therefore important to explore what implications these restrictions have for the quality of the suggestions, but this is outside the scope of this thesis.

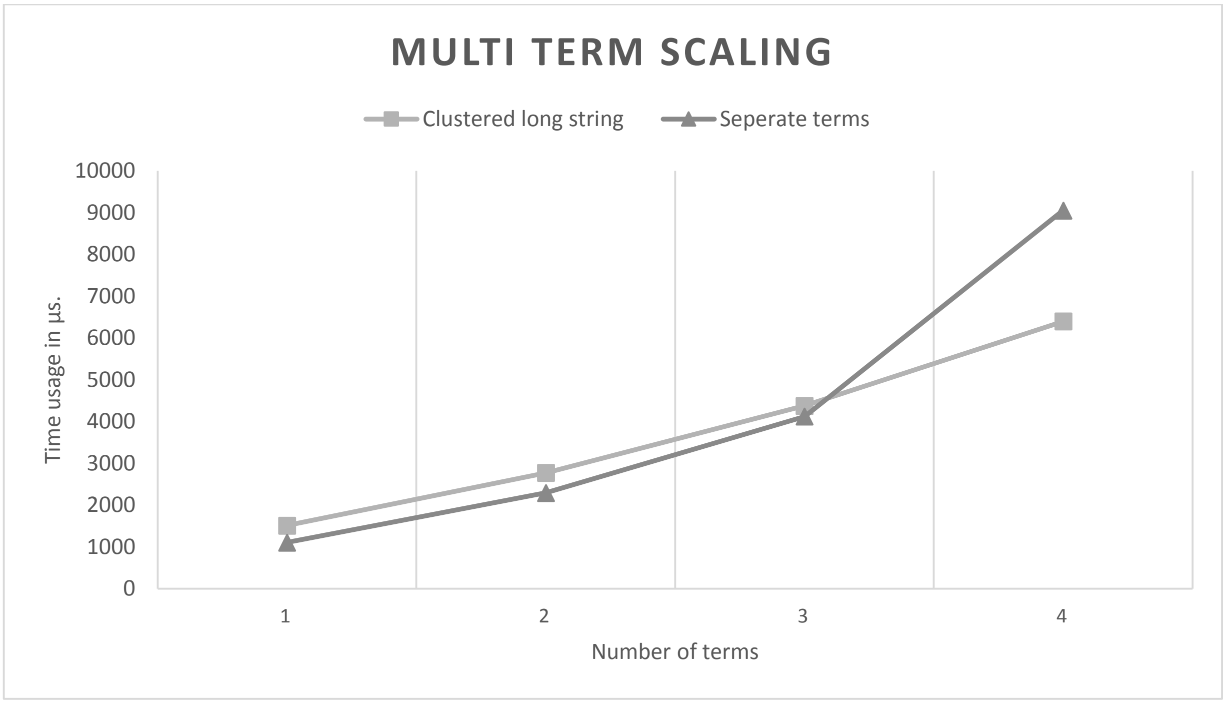


Figure 57: Results of the multi term scaling test without direct long string algorithm.

7.7 SEMANTIC CORRELATION DISCOUNT SCALING TEST

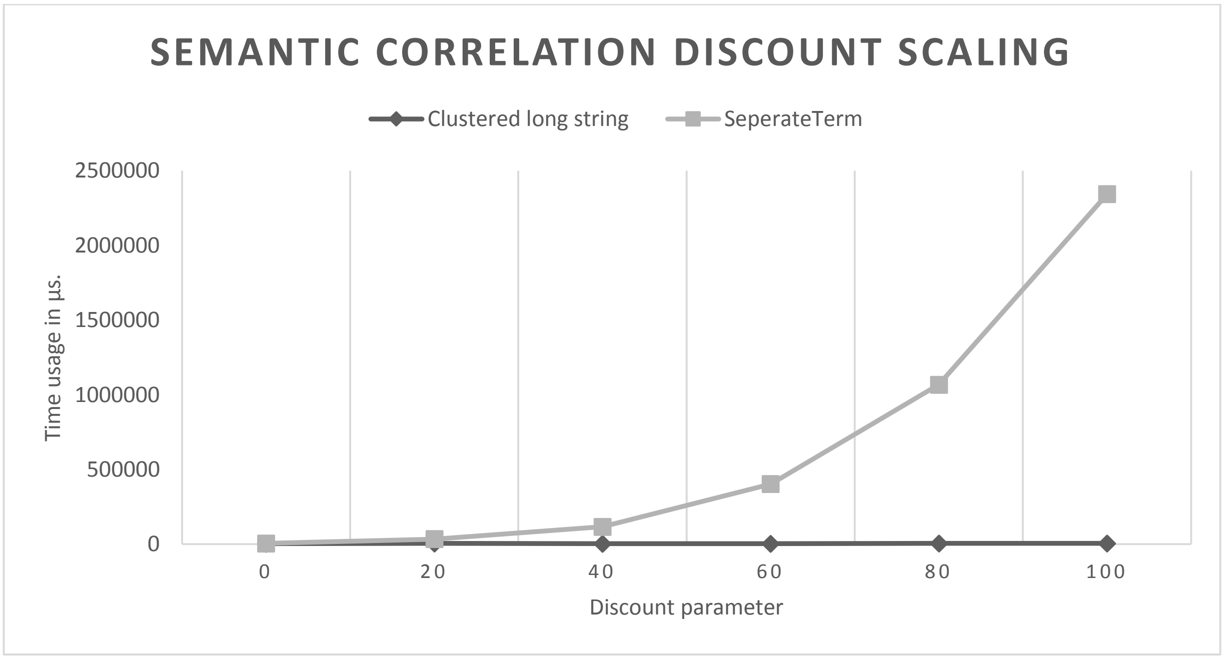


Figure 58: Semantic correlation discount scaling results.

When the semantic correlation discount increases, the direct computation of suggestion technique starts to become extremely hard. Figure 58 shows that the time usage for the separate term algorithm grows exponentially as the discount parameter increases. The direct long string algorithm were not even able to complete more than two queries within 4 hours when the parameter were 20, so we omitted the results of it, concluding that it is very slow. The clustered algorithm does however not show any clear trend of increased time usage, as we can see from Figure 59, as we could expect since it avoids the direct computation of the semantic correlation. These results shows that the direct computation is a poor option if we want high discrimination between semantically correlated terms and semantically uncorrelated terms, while the clustering technique will handle it quite gracefully.

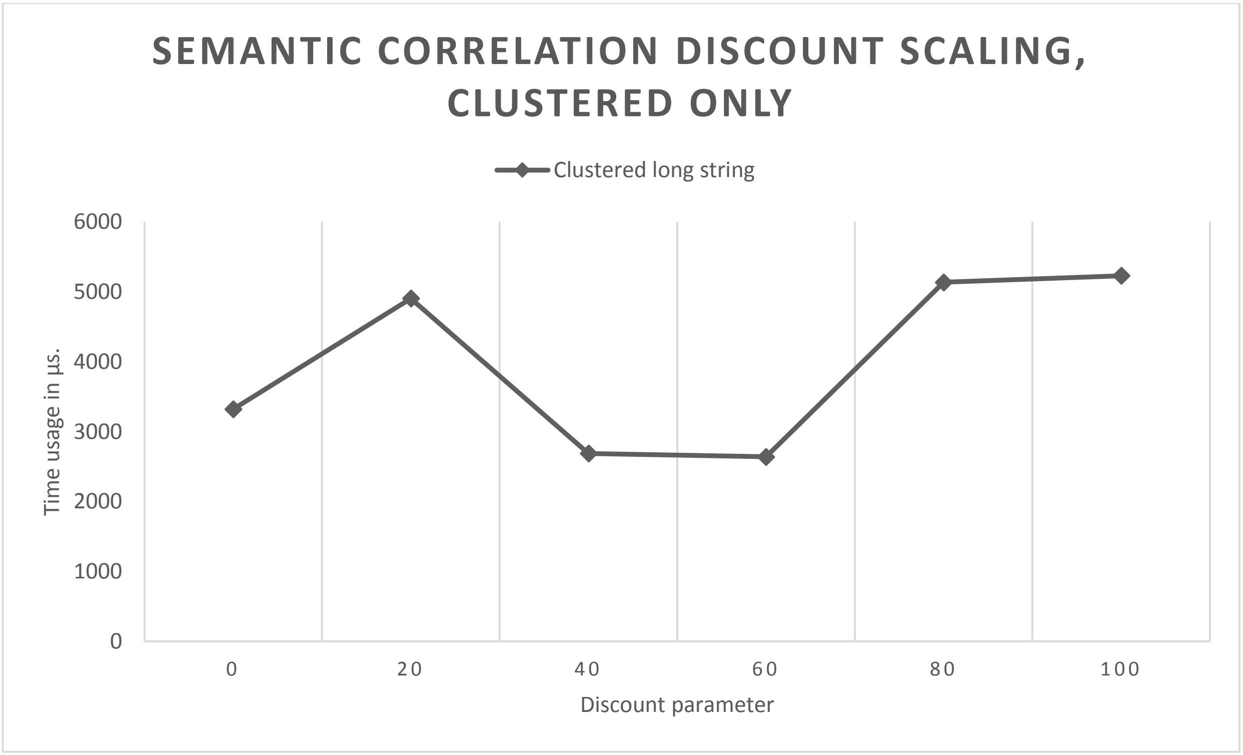


Figure 59: Semantic correlation discount scaling results, clustered only.

8 CONCLUSION AND FURTHER RESEARCH

In this study we have presented a trie based term index tailored to support efficient fuzzy interactive query expansion. We augmented the standard trie implementation with the concept of ranks for each internal nodes that represents the rank of the best descendant of that node. Further we introduced a sorted list of the children nodes of an internal trie node based on that rank. Last, we augmented the index by introducing the clustering to have different trie indexes for different bags of words where each cluster could contain terms that were semantically correlated.

We presented three interactive single term fuzzy matching algorithm that could use this index. The first was a naïve baseline algorithm, while the other two exploited our index structure in order to perform a query in a way similar to branch and bounds for the top-k single-term suggestions. We saw in our performance tests that both of these algorithms represents significant improvements over the naïve baseline. The query string based algorithm were in general faster and better scaling than the prefix based approach, except when requiring a very large amount of suggestions. Both algorithms performed in real time speed even for large indexes, clustered indexes and with a relatively large allowance for edit distance.

Multi-term queries presented us with the challenge of how to calculate semantic correlation between the terms in a multi-term suggestion without sacrificing performance. We developed three different algorithms, each with different trade-offs. We saw in our test that only two of these approaches were viable from a performance perspective. Using clusters in the index to estimate semantic cluster correlation while matching the query as a long string were really fast and did not show adverse scaling if the correlation discount were large. It does however come with the cost of a slightly larger index that is significantly harder to update. The approach were we matched the terms from the query separately were also very fast, but showed sign of scaling worse when the number of terms increased. It also get significantly slower if the correlation discount were increased.

In this study we have not explored the quality of the suggestions produced by the multi term algorithms. Exploring different ranking schemes compatible with either the cluster based algorithm or the separate terms algorithm with the focus on quality would therefore be of huge interest. The former would require looking into clustering techniques to find a good way of dividing a set of terms into semantically sound clusters. Clustering also presents challenges at indexing time. A technique for incrementally adding documents to the index without having to re-index everything would be highly desirable.

While we looked at a couple highly rewarding techniques to speed up both single and multi-term fuzzy interactive query expansion in this study, there are other techniques that could have a similar impact. Caching retrieved suggestions between each character iteration could be very interesting. We experienced very good effect of caching results for the completed query terms in our separate terms algorithm, but caching between each iteration could potentially significantly reduce the work needed to retrieve suggestions for all the algorithms we have looked at. Such technique could contribute to make other approaches like the direct long string algorithm viable.

Our approach has been restricted to a memory only implementation of the algorithms. It would be of great interest to test the performance of a system with an index persisted on disk and only partially in memory. This requires a clever way of storing the index to make the need for many nodes in various positions in the index fast, even when retrieved from slow secondary storage. Especially the combination of having an updatable index in combination with persisting it to disk might be very challenging due to the inherent difficulty of storing tree structures like a trie [26].

9 REFERENCES

[1] Y. Nemeth, B. Shapira and M. Taeib-Maimon, "Evaluation of the real and perceived value of automatic and interactive query expansion," *SIGIR,* pp. 526-527, 2004.

[2] S. Cronen-Townsend, Y. Zhou and W. B. Croft, "Predicting Query Performance," *SIGIR,* pp. 299-306, 2002.

[3] M. Magennis and C. J. v. Rijsbergen, "The potential and actual effectiveness of interactive query expansion," *SIGIR,* pp. 324-332, 1997.

[4] H. Joho, M. Sanderson and M. Beaulieu, "A Study of User Interaction with a Concept-based Interactive Query Expansion Support Tool," *ECIR,* p. 42–56, 2004.

[5] I. Ruthven, "Re-examining the potential effectiveness of interactive query expansion," *SIGIR '03,* pp. 213-220, 2003.

[6] R. W. White and G. Marchionini, "Examining the effectiveness of real-time query expansion," *Information Processing and Management,* pp. 685-704, 2007.

[7] S. Ji, G. Li, C. Li and J. Feng, "Efficient Interactive Fuzzy Keyword Search," *WWW,* pp. 371-380, 2009.

[8] S. K. Card, T. P. Moran and A. Newell, The Psychology of Human-Cpmputer Interaction, New Jersey: Erlbaum, Hillsdale, 1983.

[9] E. Ukkonen, "Algorithms for Approximate String Matching*," *Information and control,* pp. 100-118, 1985.

[10] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen and H. Li, "Context-aware query suggestion by mining click-through and session data," *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining,* pp. 875-883, 2008.

[11] E. Sadikov, J. Madhavan, L. Wang and A. Halevy, "Clustering query refinements by user intent," in *WWW '10*, 2010.

[12] Z. Liu, S. Natarajan and Y. Chen, "Query expansion based on clustered results," *Proceedings of the VLDB Endowment Volume 4 Issue 6,* pp. 350-361, 2011.

[13] J. Fan, H. Wu, G. Li and L. Zhou, "Suggestion Topic-Based Query Terms as You Type," in *International Asia-Pacific Web Conference*, 2010.

[14] B. M. Fonseca, P. Golgher, B. Pôssas, B. Ribeiro-Neto and N. Ziviani, "Concept-based interactive query expansion," *CIKM '05,* pp. 696-703, 2005.

[15] A. L. Kaczmarek, "Interactive Query Expansion With the Use of Cluster-by-Directions Algorithm," *IEEE Transactions on Industrial Electronics,* vol. 58, no. 8, pp. 3168-3173, 2011.

[16] E. Ukkonen, “Approximate string-matching over suffix trees,” *Computer Science Volume 684,* pp. 228-242, 1993.

[17] J. Wang, I. Cetindil, S. Ji, C. Li, X. Xie, G. Li and J. Feng, "Interactive and fuzzy search: a dynamic way to explore MEDLINE," *Bioinformatics 26(18),* pp. 2321-2327, 2010.

[18] G. Li, S. Ji, C. Li and J. Feng, "Efficient fuzzy full-text type-ahead search," *VLDB J. 20(4),* pp. 617-640, 2011.

[19] G. Li, J. Wang, C. Li and J. Feng, "Supporting Efficient Top-k Queries in Type-Ahead Search," Sigir, Portland, 2012.

[20] A. Behm, S. Ji, C. Li and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," ICDE, 2009.

[21] R. Baeza-Yates and B. Riberio-Neto, “Structure: Tries and Suffix Trees,” in *Modern Information retrieval*, Edinburgh, Pearson, 2011, pp. 361-362.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Radix trees," in *Introduction to Algorithms*, London, The MIT Press, 2009, pp. 304-306.

[23] Y. Sheng, Y. Li, L. Luan and L. Chen, "A Personalized Search Results Ranking Method Based on WordNet," in *Sixth International Conference on Fuzzy Systems and Knowledge Discovery*, Tianjin, 2009.

[24] C. Manning, P. Raghavan and H. Schûtze, “Introduction to Information Retrieval,” Cambridge University Press, 2009, pp. 349-375.

[25] J. Clausen, "Branch and Bound Algorithms - Principles and Examples.," Department of Computer Science, University of Copenhagen, Copenhagen, 1999.

[26] M. Barsky, U. Stege and A. Thomo, “A survey of practical algorithms for suffix tree construction in external memory,” *Softw., Pract. Exper.,* pp. 965-988, 2010.