EFFICIENT FUZZY TYPE-AHEAD SEARCH ON BIG DATA USING A RANKED TRIE DATA STRUCTURE

MASTER’S THESIS IN ENGINEERING PHYSICS
2018, John Bergman

MASTER’S THESIS IN ENGINEERING PHYSICS

SUPERVISOR: Martin Rosvall, InfoBaleen
EXAMINER: Ludvig Lizana, Umeå University
Contents

1 Introduction 9
  1.1 Background 9
  1.2 Purpose 11
  1.3 Disposition 11

2 Theory 13
  2.1 Fuzzy type-ahead search 13
    2.1.1 Previous work 13
  2.2 Measuring string similarity 14
    2.2.1 The Damerau-Levenshtein distance 14
  2.3 Prerequisites 15
    2.3.1 What is a Trie? 15

3 Methods 17
  3.1 Algorithm 17
  3.2 Preprocessing a set of records 19
  3.3 How to build our trie 20
  3.4 Extracting information from the trie 21
    3.4.1 Matching a word by traversing the trie 22
    3.4.2 Finding words within a distance k 22
    3.4.3 Auto-completion of a word 24
    3.4.4 Typos in the form of whitespaces 25
“Relevance is a search engine’s holy grail. People want results that are closely connected to their queries.”

- Marc Ostrofsky
Abstract

The efficiency of modern search engines depends on how well they present typo-corrected results to a user while typing. So-called fuzzy type-ahead search combines fuzzy string matching and search-as-you-type functionality, and creates a powerful tool for exploring indexed data. Current fuzzy type-ahead search algorithms work well on small data sets, but for big data of social networking services such as Facebook, e-commerce sites such as Amazon, or media streaming services such as YouTube, responsive fuzzy type-ahead search remains a great challenge.

This thesis describes a method that enables responsive type-ahead search combined with fuzzy string matching on big data by keeping the search time optimal for human interaction at the expense of lower accuracy for less popular records when a query contains typos. This makes the method effective for e-commerce and media services where the popularity of search terms is a result of human behaviour and thus often follow a power-law distribution.

Sammanfattning


Denna avhandling beskriver en metod som möjliggör responsiv type-ahead sök kombinerat med approximativ strängmatchning för big data genom att hålla söktiden optimal för mänsklig interaktion på bekostnad av lägre precision för mindre populär information när en sökmöteförrågan innehåller felstavningar. Detta gör metoden effektiv för e-handel och mediatjänster där populariteten av sök-terminer är ett resultat av mänskligt beteende vilket ofta följer en potens-lag distribution.
1

Introduction

1.1 Background

In the early days of e-commerce, the sales team at Amazon monitored two lists at each weekly meeting. One list for the top 10 products by popularity, and a second, more important list of the top 10 search queries that brought a customer no results. In late December, 1999, the second item on this list was "Pokeman" which is a misspelling of the famous toy brand Pokémon. The solution at that time was to re-programme the site to send the users to the correct item.¹ Today this problem is handled quite differently.

The solution is called approximate string matching or fuzzy string matching. A vital part of any search engine, and by search engine I mean a search function on an e-commerce site such as Amazon, a search function in an application such as Spotify or why not a major search engine such as Google or Bing. It is the way which one matches one string to another although they need not be equal. As you enter a search query, the search engine must match your search string to an existing one in its set of data, but people generally have misspellings and typing errors in their search query. This leads to zero results, unless there is an approximate string matching algorithm that matches the incorrect search query to the one intended.

Within e-commerce, the ability to quickly direct customers to the products that they are interested in is a key feature. Type-ahead search or search-as-you-type is a great method to achieve this. As a user types a query, matching results are presented directly at each key-stroke. Both appealing to the customer experience and also generating income as it simplifies exploring available items. Combining fuzzy string matching with the method of type-ahead search results in a very powerful tool for exploring and finding items that are relevant to a

¹Saul Hansell. Amazon’s risky christmas. New York Times (November 28, 1999), Sunday, Late Edition - Final, pages :Section 3, page 1, column 5, 1999
user. Imagine searching for a product that you only heard about, or a product where you are not sure of the spelling, or a product category for which you are not sure of how it is labeled. This would be a tedious and timeconsuming task without the combined functionality of type-ahead search and fuzzy string matching. There is a lot of research on how a user experience and react to delay and long loading times on a webpage, and a unifying result is that a user quickly loose interest and leave the page in favour of a similar site. The same result should apply to the result of searching a webpage. Let’s take Amazon.co.uk with over 250 million products as an example, and supply the string “computer memmory” to their search function. This string has one spelling error, yet the search function returns zero results on the fly. Another example is Spotify with over 30 million songs to browse. Using their search function with the query “Rock and roll all night”, which is a well known song by KISS with the exact title “Rock and roll all nite”, does not return the correct song as a result despite having over 15 million plays. Same issue when providing the query “Pnk Floyd” which refers to the artist “Pink Floyd” but having one typo. Despite having hundreds of millions of plays for all thier songs only one result is returned, an artist called ‘Pnik Floyd’ with one song available and less that 1000 plays.

A well working example is Google, which has a splendid algorithm. As you type queries, they supply so called type-ahead suggestions on-the-fly, although the query may contain many typos and misspellings and your words might be in the incorrect order. This is one of the reasons that Google is the number one search engine. One should keep in mind that they have close to unlimited access to hardware, and has probably spent a lot of resources to deal with the matter. They have large databases of their own with common typos and misspellings, all generated by user queries 2.

There is a suit of fuzzy string matching algorithms that can handle small sets of data at sufficient speed and accuracy, but large scale data requires optimized algorithms, efficient implementations, and clever filtering.

The problem becomes even more complex when combining fuzzy string matching with a type-ahead functionality, where you supply results for each key-stroke made by a user. In this case the search string is not limited to having typos, it may also be incomplete. An algorithm must be able to find relevant records although a query may be partly written: "hobbit battle of armies (The Hobbit: The Battle of the Five Armies)", contain typos: "lrod of the rongs fellowship (The Lord of the

Rings: The Fellowship of the Ring), have words in the wrong order: "judgmen day terminatr (Terminator 2: Judgment Day)". Once matching records are found, they must be presented according to relevance, which must be specifically defined. The number of typing errors is another factor of the problem, as this is a crucial parameter for performance and accuracy. Having to deal with five typos is a lot more computationally expensive than to handle two typos in a string. At the same time, the algorithm must be able to match long strings where the chance of having many errors is greater.

This project considers how to achieve fast and accurate approximate string matching together with a type-ahead search functionality on this sort of big data, where the "big data" refers to large databases with records having searchable attributes, as is often the case within e-commerce.

The above examples show that there is a need for a fast and accurate type-ahead search with fuzzy string matching functionality that can handle big data. Even large companies such as Spotify and Amazon have a very poor algorithm in use.

1.2 Purpose

The purpose of this project is to develop and implement an algorithm for combining type-ahead search with fuzzy string matching that can be applied to large datasets while maintaining a fast execution time and supply relevant results considering both the number of typos in a query string and the popularity of indexed strings. The goal is to create a search engine that can easily be implemented on existing websites and databases, targeting data that is queried and explored by people.

1.3 Disposition

This section, the Introduction, is the first out of the six that constitutes this paper. It presents a background to the problem at hand as well as the purpose of this project. Section 2, Theory, elaborates on the concept of fuzzy type-ahead search and describes previous work and its shortcomings. The measure used for string similarity, the Damerau-Levenshtein distance, is explained along with the trie data structure as they are prerequisites for the next section. Section 3, Methods, describes the algorithm for solving the problem, along with a description of all the different stages in reaching a set of matching records given a
query such as: finding similar words using the trie, how to find similar records given similar words, finding auto-complete suggestions, handle typos in the form of whitespaces, using a limited traversal to handle big data, and finally how to rank the results by relevance. Section 4, Results and Discussion, shows how the search time and accuracy is affected by the amount of indexed data, a limited trie traversal and using a popularity threshold. A discussion on the promising features of this approach and what makes it a preferred method, and also its limitations. Section 5, Conclusions, presents applicable scenarios for the method, along with possible further developments. Section 6, Acknowledgement, highlights the people making this thesis a reality.
2

Theory

The approximate string matching problem may be defined as: given a pattern string $P = p_1p_2p_3 \ldots p_m$ and a set of strings $S = \{s_1, s_2, s_3 \ldots \}$, find the strings in $S$ within a distance $d$, defined by some metric, to the pattern string $P$. This project considers an extension of this problem that involves a type-ahead functionality where the pattern string $P$ may be incomplete, meaning that it may only contain certain parts of a string in the set $S$, and these parts may not be in the correct order.

2.1 Fuzzy type-ahead search

Type-ahead search is the functionality that presents results to a user as a query is being typed, meaning that it is a method that predicts what a user is typing. It is an important tool for users when exploring data, as it provides information on-the-fly. A basic implementation of this functionality may treat the search query as one string and show all results having the exact same prefix as the query string, with an auto-complete extension of this exact prefix.

![An example of fuzzy type-ahead search.](image)

2.1.1 Previous work

Extending it to a fuzzy type-ahead search(Fig 2.1) would allow for typos in the query, additional or removed whitespaces among the query keywords as well as words in any order. Previous work\(^1\) of G.Li, Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng, Efficient fuzzy full-text type-ahead search. The VLDB Journal - The International Journal on Very Large Data Bases, 20(4):617–640, 2011

\(^1\) Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. Efficient fuzzy full-text type-ahead search. The VLDB Journal - The International Journal on Very Large Data Bases, 20(4):617–640, 2011
S.Ji, C.Li and J.Feng use a trie index structure to represent the unique words from a set of records, and matches a query keyword to prefixes in the trie, allowing typos, to finally match a record to all query keywords. Their work do however lack several important features. They do not provide the ability to apply a relevance or ranking of the results, based on for example the popularity of a record, number of typos in the query and the positional distance of the query keywords and the words in the record. Their algorithm cannot handle large scale data. They have a poor explanation of how to find the edit distance using the trie. They do not consider the edit operation transposition when calculating the edit distance nor do they handle the cases of additional or lack of whitespaces as typos in the query. This paper will propose a solution to each of these problems.

2.2 Measuring string similarity

There are several ways to define a distance or similarity between strings. They are often categorized in set-based and edit-based methods, where set-based involves constructing subsets of strings and finding similarity based on common substrings and edit-based where the number of edit operations used to transform one string into another defines the similarity between them. In this project we consider the edit-based method called the Damerau-Levenschtein distance referred to as the edit distance.

2.2.1 The Damerau-Levenschtein distance

A commonly used metric in approximate string matching is the edit distance, \( ed(s, t) \), which describes the number of edit operations needed to transform a string \( s \) to another string \( t \). The method was introduced by Damerau\(^3\) in 1964 and computationally evolved through Levenschtein, Wagner and Fischer and Ukkonen. It defines four operations that in any sequence can be used to transform one string into another. These are insertions: inserting a character ("hell" \( \rightarrow \) "hello"), deletions: deleting a character ("hellow" \( \rightarrow \) "hello"), substitutions: changing one character into another ("jello" \( \rightarrow \) "hello") and transpositions: exchanging one character to an adjacent one ("helol" \( \rightarrow \) "hello"). Each operation is assigned a cost, which is generally set to 1, but can be weighted individually for each operation. As an example I might consider it more likely that I will make a typo as a transposition rather than an insertion, meaning that I will more likely type the characters in the wrong order than actually missing a character.

There are several ways to calculate the edit distance. The common way is to use a matrix representation where the elements are calculated using dynamic programming\(^4\), but for this project we consider a different approach involving the traversal of a prefix tree, often called a trie, described in section 2.3.1.

2.3 Prerequisites

If you are unfamiliar with the abstract data type called “trie”, this section might prove useful for understanding the method section.

2.3.1 What is a Trie?

In this section we take a closer look at a specific variant of the tree data structure namely a trie\(^5\)\(^6\).

The general datastructure of a tree (Fig 2.2) can be defined as an hierarchical ordered structure of nodes, starting from a root node which in turn is linked to a set of children nodes. Each node has exactly one parent node and zero or more children. A node with no children is called a leaf. Each node has a link to its parent and also a link to each one of its children, normally directed from parent to child. A node may have a value or some other data attached to it and this is true also for a link.

A trie, also called a prefix tree, is a specific version of the tree data structure. It stores data in sequence, meaning that when data share the same sequence from their beginning, they can be represented by the same nodes. This makes the structure a great data type for storing strings. Whenever a string share a prefix with another string, they share the same nodes in the trie.

Building a trie from a set of words would look like this. For each word, start at the root. Consider the first character in the word and check if it exist among the children of the root. If it does not exist, add a new node representing that character to the children of the root. If it does exist, move down to that node. From there continue with the same procedure, looking at the next character in the string and the children of the current node, adding a new node if it does not exist else move to the existing one. When at the last character of the string, add a leaf node representing the end of the string using for example “$” as the ending character.

Figure 2.3 shows an example of building a trie. When finished, the node o, representing the prefix do, has three leafs among its de-

\(^4\) Dynamic programming is defined as breaking a complex problem into a set of simple subproblems and solve them only once, storing the solution, and iterative solve other subproblems based on previously calculated ones. In the case of edit distance, this is usually done by a matrix representation of two strings. Due to properties of this matrix the complexity of the problem can be reduced by only calculating certain diagonal elements of the matrix. These methods are called diagonal transitional algorithms and was first introduced by Ukkonen.

\(^5\) The concept of tries was introduced by Brandais and later named by Fredkin which he derived from the word “retrieval”, in “information retrieval systems”.

scendants representing the words *dog*, *dogs* and *dot*. They all share a common prefix, *do*.

Figure 2.3: Adding the words: *dog*, *dogs*, *dot*, *pie* and *pi* to a trie using $ to denote the end of a word, i.e a leaf node.
3
Methods

The problem we are facing is, given a query of one or multiple keywords, complete or partly typed, which might have typos, words in the wrong order, excluded words, lack or additions of whitespaces: find the records\(^1\) in a large set of records that is similar to the query and sort them according to how relevant they are to the query.

Before going into the details of the algorithm, the overall approach is to look at each individual keyword in the query and find similar words within an edit distance \(k\) among all the unique words that exists in the set of searchable strings. Once we have a set of similar words for each keyword in the query we define a matching string to contain one keyword from each set of similar words. If there are more than one matching string, we sort them by a relevance score that is function of the number of typos in the query, the number of exact matching keywords, the popularity of the string, the order of the keywords in the query and the string and also if the last query keyword is auto-completed.

3.1 Algorithm

The keywords in a query is treated separately since they appear incrementally as a user is typing a query. A word will appear one at a time, and the intuitive way to solve the matching problem is to consider each word. Assuming that whenever a user makes a whitespace, the preceding word is completed, and the only word that is incomplete is the last word in the query since this is the word being typed. This leads to the assumption that it is only the last word in the query that is considered for autocompletion. This incremental search form is also suitable for caching, and this is what is used for each query keyword up to a certain limit depending on the number of users using the search function. A description of the overall algorithm can be found in Algorithm

\(^1\) A record is an entry of a database. An example would be a database of movies, where each record has a field: title, genre, actors, producer, year. A suitable searchable string would be title.
A question that one might pose after this short description is “What about typos in the form of whitespaces?”. They would interfere with the assumption that we can split a query into keywords using whitespace as a separator. The answer to the question can be found in Section 3.4.4.

To handle large scale data, the algorithm limits the traversal of the trie in such a way that when finding similar word to a keyword, the popular words are prioritized. Instead of finding all possible words within a distance $k$, we find the similar commonly queried keywords at the specific time. This will enable a larger amount of records to be processed while maintaining a fast search time at the expense of accuracy on less popular records.

Algorithm 1 Overview

1. **procedure** findMatches(Query*, trie, Strings, Cache)
2. Create a list similarWords.
3. Set $n$ to the number of words in Query.
4. for each word $w_i \in$ Query do
5. Set $l$ to length of $w_i$.
6. if $w_i \in$ Cache then
7. Set similarWords$[i]$ = Cache$[w_i]$.
8. else
9. Set $k =$ nrAllowedErrors$(l)$.
10. Set similarWords$[i]$ = findWords($w_i$, trie, $k$).
11. Set Cache$[w_i]$ = similarWords$[i]$.
12. end if
13. if similarWords$[i]$ is empty and $l > 3$ then
14. hits = addWhitespaces($i$, Query, trie, Strings, Cache).
15. if hits not empty then
16. return hits
17. end if
18. end if
19. end for
20. Add auto-completed words for $w_n$ to similarWords$[n]$.
21. hits = findStrings(similarWords, Strings).
22. if hits is empty then
23. hits = removeWhitespaces(Query, trie, Strings, Cache).
24. end if
25. return hits
26. end procedure

a) Query is a list containing each word from the query. A query string “lord of the” would result in Query = [lord, of, the].

b) The number of allowed errors $k$ when finding similar words is a function of the length of the query keyword. An example would be to set $k = 1$ for $l \geq 3$ and $k = 2$ for $l \geq 5$.

c) findWords returns a list of similar words with at most $k$ typos using the trie.

d) If a keyword has no similar words in the trie, whitespaces are added at possible positions in the keyword. The algorithm for adding whitespaces is explained in Section 3.4.4.

e) Assuming that the last word in the query is the one “not finished”, a good assumption is that a user would want this and only this word auto-completed.

f) Given the list similarWords, where each element is a list containing similar words for the query keyword $w_i$, findStrings finds the strings in Strings containing a keyword from each list, and sorts them by relevance.

g) If a query has no matching strings, whitespaces are removed between keywords in the query. The algorithm for removing whitespaces is explained in Section 3.4.4.
3.2 Preprocessing a set of records

Making a set of data searchable through a fuzzy type-ahead search requires preprocessing. For each record in the data we define an item with a searchable string and a popularity, and create a list representing all these items. An element in this list may have this structure

```
{recordID: 123, string: "foo bar", popularity: 10000}
```

From the list of searchable items, we make a list of all unique words, uniqueWords, that exists in the searchable strings by separating each string into words using whitespace as a separator. Each unique word will have list of all recordID’s that contain the word, along with a list of positions of the word in that recordID. Each unique word will also have a “popularity” defined as the maximum popularity among the records containing the word. This will enable us to find the most popular similar keywords given a keyword from the query, instead of finding all words within a distance k. A unique word has the format

```
{inStrings:[1,22,456,...],
positions:[[0,3],[0],[1,4],...],
maxPopularity:13534, id:1}
```

![MovieDB](image)

**Figure 3.1:** A database of movies.

An example of the preprocessing procedure would look like this. Consider a movie database with a schema according to figure 3.1. Let the Title attribute from a record be the searchable string, and let the Views define the popularity. Two lists are generated. One list with searchable items, where each element represents a record, its searchable string, and its popularity. The second list will contain each unique word in the searchable strings, where each word has: a list with the ID’s of all the records containing the word, a list of the position(s) of
the word for each recordID, and the maximum popularity of the these records. See Algorithm 2 for a pseudo code for this example.

**Algorithm 2** Preprocessing of data

```plaintext
1: procedure preProcess(data)
2:   Create empty list searchableData
3:   Create empty list uniqueWords
4:   for each record ∈ data do
5:      Set string = record.Title.
6:      Set popularity = record.Rating.
7:      Set id = record.Id.
8:      Create new element in searchableData
9:         string = limitChars(string)
10:        words = splitToWords(string)
11:       for each w_i ∈ words do
12:          if w_i not in uniqueWords then
13:             Add w_i to uniqueWords.
14:          end if
15:          Add id to the list w_i.inRecords.
16:          Add i to list w_i.positions.
17:          Set w_i.popularity to Max(popularity, w_i.popularity).
18:       end for
19:   end for
20: end procedure
```

3.3 **How to build our trie**

The building blocks of our trie are the unique words from all the searchable strings in our set of records. Starting from the root node, each word is added character by character (Figure 2.3), either creating new nodes or sharing existing ones. Each node has a value describing the maximum popularity of the words among its descendants.
A leaf node contains the index of the word it represents in the list *uniqueWords*. Each node not being a leaf, has a list containing the indices of words among its descendents, which in turn is used for autocompletion.

### 3.4 Extracting information from the trie

Let’s start by looking at an example, again using a small set of records from a database of movies. Consider three movies: *Star Wars*, *Star Trek* and *Stargate* and let’s choose their titles to be the searchable strings and their rating to define their popularity. This leaves us with four unique words: *star*, *wars*, *trek* and *stargate* that we use to build the trie (Fig 3.2). From preprocessing we have a list, *searchableData*, where each element is linked to a record, the searchable string of that record and the popularity of that record. We also have a list, *uniqueWords*, connecting the leaves of the trie to a word. Figure 3.2 shows the trie and the data used from the preprocessing. Using this trie we can find similar words within an edit distance $k$ to a given word, by traversing the trie in a certain manner.
3.4.1 Matching a word by traversing the trie

Facing the problem of finding similar keywords to a query keyword using the trie, we define an entity of data called a queryWalker that will traverse the trie according to a keyword and maximum number of operations \( k \). The queryWalker will keep track of a few properties while traversing the trie: a current index position in the query word, the node it is currently on, the maximum number of operations it is allowed to have in its query word, and the current number of operations.

Starting with the simple case of a one-word query: stargate, without typos. We create a “walker” and place it on the root node and set its index of the query word to be the first character. We then tell it to take a step to the child corresponding to the first character in the query, in this case the child with character \( s \). We increase the index by one and perform another step to the next character in the query. We keep doing this until we end up with a leaf node, linking us to the word stargate in the list uniqueWords. The word only occurs in one string, “Stargate”, which means that we have a match for the query and can present the record from the searchableData as the result.

The next step is to handle a query with typos, to find similar words within a distance \( k \) and at the same time store the edit distance for such a similar word.

3.4.2 Finding words within a distance \( k \)

All information we need is explicitly in the trie except one important variable, the edit distance between a query keyword and a word in the trie. This, however, can be found by traversing the trie using the query keyword and a queryWalker, and at the same time allow edit operations in the form of steps to additional nodes in the trie.

Consider the query stargte, which is one insertion away from the word stargate. We create a queryWalker, place it at the root, and allow for one operation \( k = 1 \). Make it step to matching children, first to the child \( s \) and then further until its on the \( g \) node. The next character in the query is \( t \), but there is no such letter among the children. The only step the queryWalker can take is to the child \( a \), so we make that step but add one to its variable describing the number of operations. When continuing to make steps according to the letters in the query we end up at the leaf linked to the word stargate giving us a match within our
maximum number of operations \( k \). Figure 3.3 shows the path taken by the walker.

**All allowed edit operations:** insertion, deletion, substitution and transposition can be defined as a step that a queryWalker may take in the trie. A description of all steps follow below with examples from the trie in figure 3.2.

**Insertion**

Is any step to a child not being the exact matching character in the query word, while leaving the variable describing the index position unchanged. An example (Fig 3.4) would be a walker standing at the root addressing the query **targate**, having the position index of the first character in the query, and taking a step to **s**. Leaving the index position unchanged, means that at the next step we are still comparing the children with the first character in the query while the path in the trie describes the word with operations. Note that insertions in the form of the end character $ is not allowed since this correspond to one or more deletions.

**Deletion**

Is any step where the position index is increased by one, while leaving the walkers position in the trie unchanged. An example (Fig 3.5) would be a walker standing on the \( r \) node in the trie addressing the query **startgate** having an index position of 4, corresponding to the letter **t**. By increasing the index by one, the letter to consider at a next step is now **g**. Note that moving the walker to any node would render the possibility of a deletion followed by any other operation, including itself, impossible.

**Substitution**

Is any step where a walker moves to a child not being the exact matching character, and at the same time increasing the position index by one. An example (Fig 3.6) would be a walker placed at the root addressing the query **wtargate** with the index at the first character **w**, and moving to the child node with the letter **s**. The index position is increased by one. As in the case of an insertion, a substitution in the form of an end character $ is not allowed since it would correspond to one or more deletions.

**Transposition**

Is a step where the walker makes two consecutive moves, first to a child that corresponds to the character in the position: \( 1 + \) the current position index in the query word, and then to the current position index. The position index is then increased by two. This step is only possible when there are children that exactly match the
characters from the query word. An example (Fig 3.7) would be the query *tsar* and a walker standing at the root, with the position index set to the first character in the query. A first requirement is that the second character from the query, *s*, is among the children of the root, and the second is that the first character, *t*, exists among the children of the *s* node.

In order to find all similar words within a distance *k*, we must make a walker take all possible steps at each node, both exact matching steps and those that involve an operation. One walker may only take one path, which means that we have to create a new walker for each possible path. In practice, we create a new walker with the same properties as the one we are taking a step with, and add one to its number of operations, and change the position index according to the operation.

Note that several sequences of different edit operations might lead to the same node and that is why the walker with the minimum number of operations is the one to consider for a matching word. An example would be an insertion followed by a deletion, which is the same as one substitution: *starwarz* -> *starwarsz* -> *starwars* and *starwarz* -> *starwars*.

### 3.4.3 Auto-completion of a word

Each node in the trie has a list of all leafs among its descendants, i.e. the words sharing the prefix that the node represents. Auto-completing a keyword in a query is done by considering a walker having the index position on the last character in the query keyword, and not yet standing on a leaf node. Adding all leafs below the current node, i.e words sharing the prefix of that node, is to auto-complete that prefix. In our case we only consider the last word in the query for auto-completion, and also limit the use of auto-completed words for walkers with exact matching prefixes, i.e having no operations. An alternative could be to consider auto-completion for walkers with one operation, but with the restriction that the last character of the prefix must be an exact match.

Choosing auto-completed words for walkers with operations may increase the search time since it increases the number of similar words to consider quite heavily in many cases, especially if the query keyword is short. The increased number of similar words would in turn increase the number of possible strings to consider as well.

Similar words that are found through auto-completion are labeled since a later ranking of the matching strings will consider the type of the matching word when ranking the results.
3.4.4 Typos in the form of whitespaces

The method described in this paper relies on matching each individual keyword in the query by separating a query into words using whitespace as the separator. This means that a typo in the form of a whitespace must be handled in a special way when trying to find similar words to a keyword in a query.

There are two cases involving typos with whitespaces, the first is when a user exclude whitespaces between keywords in a query, typing `thegodfather` instead of `the godfather`. The second is when a user types additional whitespaces in one or more keywords of a query such as `the god father` instead of `the godfather`.

The first case is handled by separating a query keyword into two keywords whenever no similar words are found in the trie. The keyword is split at every consecutive position until the first part has an exact matching keyword in trie, and the second part has an exact matching prefix in the trie or exact matching keyword in the trie, depending on the position of keyword being split. If the keyword with no matching words is the last word in the query, the keyword is split into two and the second part of the keyword is also considered for autocompletion. An example would be a query `thegodf`, which would be split into `the godf`, and the last part of the keyword would be autocompleted as “godfather”. Note that this assumes that any one keyword in the query may only have one typo in the form of a removed whitespace. Algorithm 3 shows a pseudo code for the algorithm.

The second case is handled by joining every pair of two consecutive keywords in a query whenever no records can be matched, until a matching record is found. The reason behind this is that all the query keywords may have similar keywords in the trie, but there may not be a string containing all the keywords. In the case of the query `the god father`, each word would likely have matching words in the trie, but there wont be a string matching all separate words. Instead the first pair of words would be joined forming a new query `thegod father`. The first word in the new query would most likely not have a similar keyword in the trie, and thus no matching record can be found. The next possible pair forms the query `the godfather`, resulting in a matching record. Algorithm 4 shows a pseudo code for the algorithm.
26 efficient fuzzy type-ahead search on big data using a ranked trie data structure

Algorithm 3 Find matching strings by adding whitespaces

1: procedure ADDWHITESPACES(i, Query, trie, Strings, Cache)
2: Set l to length of wi.
3: Set isValid = false.
4: for j = 2 to (l - 1) do
5:   if wi is last word in Query then
6:     isValid = word wi(1, j) ∈ trie and
7:     prefix wi(j + 1, l) ∈ trie.
8:   else
9:     isValid = word wi(1, j) ∈ trie and
10:    word wi(j + 1, l) ∈ trie.
11: end if
12: if isValid then
13:   Set Query = Query(1, i - 1) + wi(1, j) +
14:     wi(j + 1, l) + Query(i, end)\(^a\).
15:   hits = findMatches(Query, trie, Strings, Cache).\(^b\)
16:   return hits
17: end if
18: end for
19: return Empty list.
20: end procedure

Note that the second case could generate a first case scenario. If two keywords are joined and no matching word is found then that would trigger a first case scenario, as in the example above when the keyword thegod is formed. That is why the two cases must be separated in the matching algorithm. When trying to match a query, first consider the case when a query keyword has no matching words in the trie. Find matching words by splitting the keyword at every consecutive position. If there still are no matching words, consider the second case and join consecutive pair of keywords until a matching record is found.

a) The query is changed to include the new words. An example would be the query, Query = \([\text{lord, of, the, rings}]\), where the second word would be split, forming the new query Query = \([\text{lord, of, the, rings}]\).

b) The findMatches function is called with the new query where the keyword wi has been split into two. The first part has an exact matching word in the trie and the second has either an exact matching word in the trie or an exact matching prefix in the trie in the case of wi being the last word in the query. Note that by calling the function recursively we are able to handle multiple typos involving whitespaces. Another word after wi in the query may also have lack of whitespaces.
Algorithm 4 Find matching strings by removing whitespaces

1: procedure removeWhitespaces(Query, trie, Strings, Cache)
2:   for i = 1 to n do
3:     Set newQuery = Query(1, i - 1) + join(w_i, w_{i+1}) +
4:     Query(i + 2, n).
5:   for w_j ∈ newQuery do
6:     if w_j ∈ Cache then
7:       Set similarWords[j] = Cache[w_j].
8:     else
9:       Set similarWords[j] = findWords(w_j, trie, k)
10:   end if
11: end for
12: hits = findStrings(similarWords, Strings).
13: if hits not empty then
14:   return hits
15: end if
16: end for
17: return Empty list.
18: end procedure

3.5 Handle Big-data

Performing type-ahead search on a very large set of data can decrease the performance heavily when allowing errors in a query. Dealing with a set of strings in the order of hundreds of millions will be too computationally expensive to have results within a viable search time of type-ahead search. Figure 3.8 shows popular services and sites having this problem.

A limited traversal for walkers in the trie solves this problem, with a tradeoff in accuracy. The idea is to find the more popular similar words for a query keyword at a given time. Popularity is, per definition, describing the most frequently searched records, and is something that changes over time. By limiting the traversal in this way, the fuzzy type-ahead search is always able to match the most queried records, while keeping a viable search time. This method has one drawback, the accuracy of words having low popularity will decrease, meaning that the records with low popularity would lose the ability to be found with typing errors. Instead the algorithm will maintain a fast search time for the more popular words since all possible words within a distance k of a query keyword wont be considered when find-
ing matching records. This may not be of such a large issue, especially when considering e-commerce, as the more popular words among the records are the words that are most frequently typed by any user.

The complexity of performing the different operations in the trie is dominated by insertions and substitutions, since a deletion and a transposition at character in a query only creates one new walker per operation while an insertion and a substitution creates a maximum amount of walkers corresponding to the allowed number of characters in the alphabet used since that is the maximum number of children at any node.

In practice, the traversal is limited in the following way: When a walker is asked to take a step, which in turn creates a new walker for each possible operation from the current node, the steps representing the insertion and substitution operation is limited to nodes having a maximum popularity among its descendants above a certain threshold $T$. For example we can set the threshold to correspond to 10% less than the maximum popularity, meaning that the top 10% similar words regarding popularity would be prioritized. Algorithm 5 shows how the traversal is performed when finding similar words in the trie.

3.6 How to find a matching record

Assume a query with multiple keywords $Q = \{w_1, w_2, \ldots, w_l\}$ where each keyword $w_i$ has a set of similar words $K_i = \{k_{i1}, k_{i2}, \ldots\}$ within an edit distance $d$. Each keyword $k_{ij}$ has an inverted list $L_{ij} = L(k_{ij})$ with all record ID’s containing the keyword. We want to find the record or records that contains a keyword from each set $K_i$ for $i = 1, 2, \ldots, l$ since we require each keyword in the query to exist in a matching record. Let $U_i = \bigcup_{k_{ij} \in K_i} L(k_{ij})$ be the union of all inverted list of a keyword $w_i$. The intersection of the unions $\bigcap_{1 \leq i \leq l} U_i$ corresponds to all records matching the query $Q$.

3.6.1 Finding similar words to a query keyword

The first part of finding the results to a query $Q$ is to consider each keyword $w_i$. By combining the trie with the traversal of queryWalkers and performing steps according to a limited traversal we find the similar words within an edit distance $k$ (Algorithm 5).

This results in a list of similar words within an edit distance $k$.\[\text{\textit{}}}
Algorithm 5 Find similar keywords in the trie.

1: procedure findWords($w_i$, trie, $k$)
2:     Create empty list similarWords
3:     Create empty priority queue Queue
4:     Create a queryWalker $qw$ at root node with $maxOperations = k$,
5:     operations = 0 and index = 0.
6:     Add $qw$ to Queue.
7:     while Queue not empty do
8:         Dequeue a walker $qw$.
9:             repeat
10:                 if Number of operations of $qw < k$ then
11:                     Create a new walker for each possible step that
12:                     requires an operation from current state of $qw$,
13:                     where a new node has a descendant popularity > $T$
14:                     for insertions and substitutions and place each
15:                     one in Queue.
16:                 end if
17:                 Move $qw$ to the node that corresponds to exact
18:                 matching character in $w_i$.
19:             until No exact step is possible or $qw$ on a Leaf
20:             if $qw$ on Leaf then
21:                 Add $qw$ to similarWords.
22:         end if
23:     end while
24:     return similarWords
25: end procedure
3.6.2 Using a cache to increase performance

A naive approach of handling a query would be to perform a search each time a user enters a character in the search field, without taking the results from the previous query sent for the previously entered character. If we instead use a cache to save the list of similar words given a query keyword, we only need to traverse the trie and find similar words for the last keyword in the query. This makes the part of the search time that involves finding similar words independent of the number of keywords in the query.

Whenever a whitespace is the last character typed by a user, the preceding query keyword and its set of similar words is added to the list acting as the cache. When a new query is received, i.e., when additional characters are typed in the search field, each query keyword is looked up in the cache list. If it exists, the set of similar words for that keyword is appended to the list of similarWords, else a new set of similar words are found using the trie. There is also the case of a user pasting a query into the search field, in which the cache probably won't contain the words, unless a previous user has typed a similar query.

3.6.3 Matching a multiple words query

To find the intersection \( \bigcap_{1 \leq i \leq l} U_i \) of the unions of each keyword \( w_i \) in an effective way, we only consider the shortest union \( U_m \) among the words in the query. For each record in this shortest union we create five attributes that will enable a later ranking of the matching records: the number of matching keywords, the number of exact matching keywords, the sum of edit operations for the keywords matching the record, a boolean representing the fact that the last query keyword is auto-completed, a list with the position of each matched keyword in the searchable string, and check the other unions if they contain that record. If it is we add one to the number of matched keyword in the record, and depending on the attributes from the keyword matched in the query, we also add the number of operations, add one to the number of exact matched keywords, the position of the keyword in the searchable string of the record, and if it is an autocompleted keyword. When all unions are processed, the matching records for the query \( Q \) are the ones having number of matched keywords = 1. Algorithm 6 shows a pseudo code for the algorithm. Figure 4.1 shows a small example of a query with three keywords that has been matched to a set of records.
The complexity of this procedure when regarding time is $O((l - 1) \sum_{i \neq m} U_i)$ where $U_m$ is the smallest of the unions, which implies that the more keywords there are in the query the more unions there are to process, resulting in a longer search time. Note that by using a hash function for indexing the records in the smallest union list, the look-up time for a record ID is a constant $O(1)$. The space complexity is $O(U_m)$ since the only union that is materialized is the smallest one, along with the attributes for each record in the union.

### 3.7 Ranking the results

The most important feature of any search engine is to supply relevant results. Once we have a set of matching records, they must be presented according to the relevance of the query supplied by the user. The information available to us for defining a relevance for a record are: the popularity, the total number of operations for the matched keywords, the number of exact matching keywords, the positional distance of the keywords in the record and in the query, and finally the fact if the last keyword has been autocompleted. How to decide the importance of each attribute might depend on the data being queried, but I will present one way to order them and argue why this is a good idea.

#### 3.7.1 Defining a proximity metric between strings

One parameter for defining the relevance of a record is how similar the position of the keywords in the query are to the keywords in the string.
Algorithm 6 Find matching strings.

1: procedure FINDSTRINGS(similarWords\textsuperscript{a}, Strings)
2: \hspace{3em} Create an empty list hits.
3: \hspace{3em} Create a list, possibleStrings\textsuperscript{b}, with all recordID's from the
4: \hspace{3em} smallest union \( U_m \) among the elements of similarWords.
5: \hspace{3em} for union \( U_i, i \neq m \) from similarKeywords do
6: \hspace{6em} for each similar word \( w_j \) in \( U_i \) do
7: \hspace{9em} for each recordID in the inverted list \( L_{ij} \) do
8: \hspace{12em} if recordID exists in possibleStrings then
9: \hspace{15em} if \( w_j \) contains no edit operations then
10: \hspace{18em} Add 1 to \( nr\text{ExactMatches} \).
11: \hspace{15em} else
12: \hspace{18em} Add number of operations of \( w_j \) to
13: \hspace{18em} \( sum\text{Operations} \) for record having recordID.
14: \hspace{15em} end if
15: \hspace{12em} Add 1 to \( nr\text{Matches} \).
16: \hspace{12em} Add the positions of \( w_j \) in the string of recordID
17: \hspace{12em} to the list positionsInString.
18: \hspace{9em} end if
19: \hspace{6em} end for
20: \hspace{3em} end for
21: \hspace{3em} for each element \( e \) in possibleStrings do
22: \hspace{6em} if \( nr\text{Matches} = \) Number of keywords in query then
23: \hspace{9em} Add \( e \) to hits.
24: \hspace{6em} end if
25: \hspace{6em} end for
26: \hspace{3em} end for
27: \hspace{3em} return hits
28: end procedure
That is why we define a proximity metric that will tell the positional distance between the query and the matching record.

**Each matching string** has a list describing the positions of the matched keywords in that string. If a word is found in more than one position, that element in the list is itself a list containing each position of that keyword. This allows for multiple combinations for positions of the matched keywords. An example of this can be seen in figure 4.1 for the string “The Lord of the Rings: The Fellowship of the Ring” and the query “lodr of the” when the keywords “the” and “of” occur at several positions in the string. This results in a set of possible combinations of positions that has to be evaluated, in this case: \{[1,2,0], [1,2,3], [1,2,5], \ldots, [1,7,8]\}.

The positional distance is calculated by considering the relative difference of the position in the query and the position in the string. The list of positions for each string is normalized to correspond to the relative difference to the position of the first matched keyword. As an example, the combination [1,2,0] from the string “The Lord of the Rings: The Fellowship of the Ring” would be changed to [0,1,−1] in order to reflect the relative positional distance to the first matched keyword. Taking the Euclidean distance\(^3\) of the the query positions \([0,1,2]\) and the positions of the keywords \([0,1,−1]\) gives the positional distance \(d = \sqrt{(-1-2)^2} = 3\). In practice, the positional distance is evaluated for each possible combination of the matched keywords in a record, and the minimum is stored as one of the parameters used in finding the relevance of a record.

If two or more keywords are identical in the query, the combinations generated might be invalid since one keyword must not be allowed at two positions in a string. An example would be the query **the lord of the** matched to the string “The Lord of the Rings: The Fellowship of the Ring”. The positions list \{[0,3,5,8],1,2,[0,3,5,8]\} would have combinations such as \([0,1,2,0],[3,1,2,3]\)\ldots\ which would correspond to the first occurrence of the keyword “the” from the query being matched at two positions in the string. The issue is solved by removing all combinations where a position occurs twice or more.

Note that this distance does not take into account where the keywords are in the string. The query keywords **lord of the** could be matched to the beginning of a string “Lord of the \ldots” or at the end of a string “… lord of the \ldots” with hundred words prior to the matched keywords and end up with the same positional distance.

\(^3\) Given two points in Euclidean n-space, \(p = [p_1, p_2, \ldots, p_n]\) and \(q = [q_1, q_2, \ldots, q_n]\), the Euclidean distance is calculated as \(\sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}\).
3.7.2 Defining the relevance of matching records

The relevance of each matching record is calculated using the properties of the matched keywords, and may be defined in any way one sees fit. Here we define a score that we believe has a beneficial ordering, and that is calculated as a linear combination of five variables $f(x_1, x_2, x_3, x_4, x_5)$, where $x_1 = \text{number of exact matching keywords}$, $x_2 = \text{popularity of record}$, $x_3 = 1$ if last keyword is auto-completed, else 0, $x_4 = \text{relative distance of matched words}$, $x_5 = \text{sum of operations for matched words}$.

How do we want the relevance to work? Consider a case where a matching record has one typo and a very low popularity, and another matching record has two typos but has a very high popularity. Which one is more relevant to a user? A good assumption is that a user supplies a correct query. This leads us to the following rules.

The linear combination is weighted to achieve the ordering as follows

1. **Number of exact matching keywords.**
   A matched record with $M$ exact matching keywords and a total of $O$ operations should be more relevant to a user than a matched record with less than $M$ exact matching keywords and with any number of operations. Note that there is a limit to the number of operations allowed in a search string and word.
   Example: Consider a query with three keywords. A record matching two keywords with no operations, and one keyword with two operations, should have a higher relevance than a record matching one keyword with no operations and two keywords with one operation each.

2. **Non auto-completed last keyword.**
   Matching records where the last keyword in the query has not been auto completed should be more relevant than a matching record where any auto completed word has been used as long as the last query keyword does not have any operations without auto completion.
   Example: Consider a query having two keywords. A record matching the first keyword with one operation and the second with no operations should have higher relevance than a record matching the first keyword with one operation and the second keyword with no operations but being autocompleted.

3. **Total number of operations**
   A matching record with $O$ or less operations should have higher
relevance than a matching record with more than $O$ operations.

Example: Consider a query consisting of two keywords. A record matching one keyword with one operation and the other keyword with one operation, has higher relevance than a record matching the first keyword with one operation and the other keyword with two operations.

4. **Relative distance of keywords in query and record.**

Consider a query with two keywords. A matching record where both keywords match and are in the same order in the query as in the record should be more relevant than a matching record with both keywords matching but where the keywords are not in the same order in the record as in the query.

5. **Popularity of record.** When matching records are sorted according to the above rules and results have equal relevance, the popularity of the record should decide the relevance ordering.

If there are records tied at any numbered property, we look at the next property until each record has a unique relevance. The ordering of the parameters is based on the assumption that whatever a user is typing is correct. Thus the first priority is to consider the number of matching keywords with no operations. If that alone can't separate any candidate records, then the second is to consider whether the last keyword in the query is matched using auto-completion. If there still is a tie in the relevance, the third is to consider the total number of operations of the matching records. If still tied, we consider the fourth which is the relative distance in the query and in the matching records, and prioritize according to lower distance. And finally we consider the popularity of the matching records.
4

Results and Discussion

We used three different relations to evaluate our algorithm, how the search time depends on the size of indexed data, how the number of operations required for finding similar word varies with a limited traversal and without, and how the accuracy varies with the popularity of a query. The data used was a set of titles from Wikipedia\textsuperscript{1} containing 3 million records. The popularity of the titles was set to range from 0 to 1, and distributed according to a power law $\text{Popularity} \propto x^{-2.5}$, since popularity generated by human behaviour often follow the “rich-get-richer” phenomenon\textsuperscript{2}.

We choose to divide the span of popularity into 10 intervals, and randomly selected 300 random queries from the indexed data for each interval. To have the results focus on the matching of words, and not involve finding the intersection of several sets of similar words, a query was limited to having only one keyword. The length of a query was limited to 5-15 character, and the queries were randomly distributed over string length for each interval. Each string was transformed to having 1-2 typos, randomly chosen among the four edit operations, where one typo was allowed for words with a minimum length of 3 characters, and two typos was allowed for words with a minimum length of 6 characters. The traversal threshold $T$ is limited to 10% of the maximum popularity of all indexed records. For a search to be considered as successful the expected record must be within the top 10 returned records.

4.1 Using limited traversal

The results from using a limited traversal according to a popularity threshold on insertions and substitutions has the expected effect. The accuracy is 100% for the queries in the interval with top 10% popularity, and falls off to around 50% for the least popular queries. These mostly account for the queries with typos in the form of deletions and

\textsuperscript{1} Up-to-date database dumps of Wikipedia can be found at: http://dumps.wikimedia.org.

\textsuperscript{2} David Easley and Jon Kleinberg, Networks, crowds, and markets: Reasoning about a highly connected world. Cambridge University Press, 2010
transpositions as we do not limit these operations in the traversal of the trie.

Note that the accuracy need not be 100% although every possible similar word within distance \( k = 1, 2 \), at the lengths specified above, are considered for every keyword in the query. This is because for some queries there might be more than 10 matching records having less distance than the record originally transformed with \( k \) edit operations. Figure 4.2 shows an example where applying three edit operations, an insertion and two substitutions to a query \texttt{star wars}, generating the query string \texttt{start cart}, which has 10 completely separated results from the query \texttt{star wars}.

If we look at the search time and compare a complete traversal of the trie to a limited one, figure 4.3, we find that it has little effect when handling queries with one typo while two typo queries is a lot more prominent. This is because handling two typos is a lot more computationally expensive. The limited traversal is on average two times faster than a complete traversal when handling two typos.

The goal of having a search engine where the search time is independent of the amount of indexed data is not at hand. But using a threshold rule for the top 10% as in this case, is a suitable compromise that provides a fuzzy type-ahead search functionality to a data set queried by humans generating power law distributions and with constantly changing popularity, always providing relevant results.
Figure 4.3: The search time is reduced to half or more for queries having two typos.
5
Conclusions

The methods described in this paper results in a great way to provide a modern day search engine on ever changing data and where access and exploration is of human nature.

We provide a way to index large sets of data using a trie data structure, that can be queried with all forms of typos: insertions, deletions, substitutions and transpositions including whitespaces. We define all available edit operations as steps traversing the trie, and we provide a way to find similar words to a query word using these steps. From similar words for each query word, we find similar records that we present as results. We provide results that are relevant by defining a relevance scoring using information about number of exact matching keywords, auto-completion, number of typos, relative distance of query keywords and indexed records and popularity. All this while keeping the search time viable for human interaction.

5.1 Further development

There are a lot of implementation enhancements that would improve the performance of the search engine as the implementation was not the main focus of this work. Considering the method, there are some ideas that were not given time to explore during this thesis.

A probability based method for the trie traversal is something that was touched briefly on during implementation of the search engine. Using a probability defining the likelyhood of each prefix given the query, combined with actual probabilities for each edit operation and traverse the trie according to the most probable paths would allow for an even more optimized traversal of the trie.
A dynamic cost for edit operations could be implemented with little effort. If one had statistics on the occurrences of the different operations whenever a typo occurs, then this would improve the relevance of a match. A matching record where the typos are more commonly occurring would have a higher score and thus a higher relevance. Take into account the probability of different edit operation and weight the cost of the operations accordingly. This could be implemented as a self learning trait of the search engine.
6 Acknowledgement

I would like to thank everyone at Infobaleen, especially my supervisor Martin Rosvall and Christian Persson, who has supported me throughout this project, pointed me in the right direction and given me loads of inspiration and ideas. Also, I would like to thank the people at IceLab for the creative environment and supporting atmosphere they provide.
Bibliography


