# Approximate Dictionary Searching at a Scale using Ternary Search Trees and Implicit Levenshtein Automata

Peter Schneider-Kamp[a]
*Department of Mathematics and Computer Science,*
*University of Southern Denmark, Campusvej 55, Odense, Denmark*

Keywords: Approximate Dictionary Searching, Ternary Search Tree, Edit Distance, Levenshtein Automata, Scalable Algorithms.

Abstract: Approximate Dictionary Searching refers to the problem of finding entries in a dictionary that match a search word either exactly or with a certain allowed distance between entry and search word. Extant computationally efficient data structures and algorithms addressing this problem typically do not scale well to large alphabets and/or dictionaries, often requiring prohibitive amounts of memory as the sizes of alphabets and dictionaries increase. This paper presents a data structure and an algorithm for approximate dictionary searching that rely on ternary search trees and implicit Levenshtein automata and scale well with the sizes of both alphabets and dictionaries.

## 1 INTRODUCTION

Approximate dictionary searching refers to the problem of finding entries in a dictionary that match a search word. Given a search string $k$, a distance measure $\delta$ between two strings, and a distance threshold $t$, the task is to find all entries $e$ of the dictionary such that $\delta(k,e) \leq t$. When $\delta(k,e)$ is zero, $e$ is an exact match for $k$. Otherwise, it is an approximate match.

Many data structures and algorithms addressing the approximate dictionary search problem have been proposed. Boytsov (Boytsov, 2011) reviews and empirically compares the most prolific data structures and algorithms, finding that many of these are challenged by the size of alphabets and/or the size of dictionaries. In the experiments, the largest alphabet size considered is 36 and the largest dictionary has 3.2 million entries.

Tries (aka prefix trees) as data structures with a search algorithm based on explicit Levenshtein automata arguably constitute one of the best performing and elegant solutions to approximate dictionary searching. A trie for an alphabet of size 36 holding 3.2 million entries holds at least 1.152 billion pointers. This is a lower limit under the (unrealistic) best-case assumption that all the entries only differ from each other by exactly one character. In this optimistic case and assuming a 64-bit architecture, the trie would consume "only" approx. 8.5 GByte of memory.

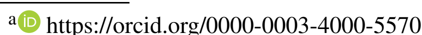
This scaling behaviour makes the otherwise elegant and efficient trie-based solution prohibitive for larger real-world scenarios. When consulting a company offering digital dictionary services, an alphabet of size 7040 for a dictionary with 162.193.908 entries had to be stored and searched efficiently to support up to 10.000 simultaneous users. The memory consumption of a trie-based solution was estimated to be at least 10 TByte.

Ternary search trees are a slightly less computationally efficient but more memory efficient alternative to tries. This paper presents a data structure and an algorithm for approximate dictionary searching that rely on ternary search trees and implicit Levenshtein automata and scale well with the sizes of both alphabets and dictionaries both regarding runtime and memory usage.

The main contributions of this paper are:

1. The use of ternary search trees for approximate dictionary searching.
2. A search algorithm that uses implicit Levenshtein automata.
3. A formal proof that the proposed solution is sound and complete.

The remainder of this paper is structured as follows. Section 2 concisely reviews the necessary background and related work. Section 3 introduces the proposed solution for approximate dictionary search-



[a] https://orcid.org/0000-0003-4000-5570



Schneider-Kamp, P.
Approximate Dictionary Searching at a Scale using Ternary Search Trees and Implicit Levenshtein Automata.
DOI: 10.5220/0011312300003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 657-662
ISBN: 978-989-758-588-3; ISSN: 2184-2833
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Algorithm 1: Adding entries to a ternary search tree.

```
procedure ADD(n,s,e)
    if n = UNDEF then
        n.c ← HD(s)
    end if
    if HD(s) < n.c then
        n.ℓ ← ADD(n.ℓ,s)
    else if HD(s) > n.c then
        n.r ← ADD(n.r,s)
    else if |s| > 1 then
        n.r ← ADD(n.m,TL(s))
    else
        n.e ← e
    end if
    return n
end procedure
```

ing. Section 4 evaluates this solution empirically before Section 5 briefly concludes.

## 2 BACKGROUND

A ternary search tree (Bentley and Sedgewick, 1997) is a search tree where each node has three children nodes: a left node leading to lexicographically smaller entries, a middle node leading to longer entries with the exact prefix, and a right node leading to lexicographically larger entries. This paper defines a ternary search tree over a given alphabet $\Sigma$ and a distance measure $\delta$ as a triple $\langle \Sigma, \delta, \rho, \mathcal{N} \rangle$ where $\mathcal{N}$ is a set of nodes and $\rho \in \mathcal{N}$ is the root. Each node $n$ is a quintuple $\langle c, v, \ell, m, r \rangle$ where $c \in \Sigma$ is the next letter of the entry, $v$ is the value associated with the entry, and $\ell = \langle c_\ell, \ldots \rangle, m = \langle c_m, \ldots \rangle, r = \langle c_r \rangle \in \mathcal{N}$ such that $c_\ell < c_m < c_r$.

Adding entries to a ternary search tree is straightforward by traversing the tree from the root. By executing ADD($\rho$,$s$) as described in Algorithm 1, the string $s$ is added. Checking for an exact match with the string $s$ follows the same pattern and is described in Algorithm 2. In both cases, HD and TL are functions that return the first letter of a string and the remaining string, respectively. For example, HD("Levenshtein") = "L" while TL("Levenshtein") = "evenshtein".

The Levenshtein distance (Levenshtein, 1966) between two strings $s_1, s_2 \in \Sigma^*$ is the minimal number of edits (insertions, deletions, and replacements) needed to make these two strings identical. For the following definition, without loss of generality, we assume $|s_1| < |s_2|$:

Algorithm 2: Exact search in a ternary search tree.

```
procedure GET(n,s)
    if n = UNDEF then
        return UNDEF
    end if
    if HD(s) < n.c then
        return GET(n.ℓ,s)
    else if HD(s) > n.c then
        return GET(n.r,s)
    else if |s| > 1 then
        return GET(n.m,TL(s))
    else
        return ⟨n.e,n.v⟩
    end if
end procedure
```

$$\mathrm{LEV}(s_1,s_2) = \begin{cases} |s_2| & \text{if } |s_1| = 0 \\ \mathrm{LEV}(\mathrm{TL}(s_1),\mathrm{TL}(s_2)) & \text{if } \mathrm{TL}(s_1) = \mathrm{HD}(s_2) \\ 1 + min \begin{cases} \mathrm{LEV}(\mathrm{TL}(a), b) \\ \mathrm{LEV}(a, \mathrm{TL}(b)) \\ \mathrm{LEV}(\mathrm{TL}(a), \mathrm{TL}(b)) \end{cases} & \text{otherwise} \end{cases}$$

A fast way of checking that the edit distance between $s_1$ and $s_2$ is below a threshold $t$ is to construct a Levenshtein automaton (Schulz and Mihov, 2002) for distance $t$ and string $s_1$. The automaton is built in such a way that it accepts all strings $s \in \Sigma^*$ such that $\delta(s_1, s) \leq t$. The size of Levenshtein automata grows with the threshold $t$, the size of the string $s_1$ and the alphabet $\Sigma$.

Approximate dictionary searching is based on approximate string matching, for which Ukkonen presented an efficient algorithm (Ukkonen, 1985). For an overview of string matching, see Navarro's guide (Navarro, 2001). Different approaches to approximate dictionary searching have been reviewed systematically and evaluated empirically (Boytsov, 2011), including tries with Levenshtein automata. Leveling et al. (Leveling et al., 2012) mention the use of ternary search trees without providing details of their implementation of approximate matching.

## 3 TST FOR APPROXIMATE DICTIONARY SEARCHING

The main idea of this paper is to store all the dictionary entries in a ternary search tree and use implicit compressed Levenshtein automata to implement an efficient search algorithm for approximate matches.

In other words, while following the general idea of Algorithm 2, we keep track of the remaining number of edits we are allowed to perform. In the search through parts of the tree, as long as we still have at

least one edit that we are allowed to perform, we descend into both left, middle, and right children.

When considering a possible deletion, the search continues with the first letter of the search word removed. When considering a possible insertion, the search continues with a wildcard flag $\omega \notin \Sigma$ that signifies that the first letter of the search word now matches all possible letters from $\Sigma$. Likewise, when considering a possible replacement, the search continues with a search word where the first letter has been replaced by a wildcard flag $\omega \notin \Sigma$. For example, if the search word is “evenshtein”, the search continues with “venshtein” for deletion, “ωevenshetin” for insertion, and “ωvenshtein” for replacement.

The use of a wildcard flag and threshold corresponds to a compressed Levenshtein automaton. In order to prune redundant paths through the tree, the algorithm also keeps track of whether possible edits already have been explored for this part of the path.

The approximate search algorithm is presented in Algorithm 3. Here, n is the current node under consideration, s is the search word, t is the threshold, v is the value associated with an entry to possibly return as a result, w is a flag indicating that a wild card precedes the search word, and d is a flag indicating that the current part of the path has already been explored in relation to possible edits. The final parameter keeps track of the entry such that entries and values can be returned as pairs. We denote the empty word as $\varepsilon$ and string concatenation as an infix operator $\cdot$. The algorithm returns a list of results consisting of pairs of entries and associated values.

For the sake of clarity, the construction of this list is made implicit by the **yield** and **yield from** statement implementing the popular generator semantics of high-level languages such as Python. Here, **yield** adds a single value to the implicit result list while **yield from** adds all values from the implicit result list of a recursive call.

The presented algorithm for approximate search in ternary search trees is guaranteed to find all exact matches, as well as matches that require at most $t$ edits from the search word. Likewise, it is guaranteed not to find any other matches.

Algorithm 3: Approximate search in a ternary search tree.

```
procedure GET(n, s, t)
    yield from _GET(n, s, t, UNDEF, FALSE, FALSE, ε)
end procedure
procedure _GET(n, s, t, v, w, d, e)
    if |s| = 0  and ¬w  and v ≠ UNDEF then
        yield ⟨e,v⟩
    end if
    if n ≠ UNDEF  and (|s| ≥ 1  or w) then
        if w  or HD(s) < n.c then
            yield from _GET(n.ℓ, s, t, UNDEF, w, TRUE, e)
        end if
        if w  or HD(s) > n.c then
            yield from _GET(n.r, s, t, UNDEF, w, TRUE, e)
        end if
        if w  or HD(s) = n.c then                                    ▷ consume letter
            yield from _GET(n.m, (w ? s : TL(s)), t, n.v, FALSE, FALSE, n.c · e)
        end if
    end if
    if ¬c  and t ≥ 1  and ¬w then                                    ▷ edit allowed
        if n ≠ UNDEF then
            yield from _GET(n, s, t-1, UNDEF, TRUE, FALSE, e)         ▷ insert
        end if
        if |s| ≥ 1 then
            yield from _GET(n, TL(s), t-1, UNDEF, FALSE, FALSE, e)    ▷ delete
        end if
        if n ≠ UNDEF  and |s| ≥ 1 then
            yield from _GET(n, TL(s), t-1, UNDEF, TRUE, FALSE, e)     ▷ replace
        end if
    end if
end procedure
```

**Lemma 1** (Soundness and Completeness)**.** *Let $\mathcal{T} = \langle\Sigma, LEV, \rho, \mathcal{N}\rangle$ be a ternary search tree, $s \in \Sigma^*$ be a string, and $t \geq 0$ be an integer.*

*Let $\mathcal{E} = \{e \in \Sigma^* \mid GET(\rho,e) = \langle e,v\rangle$ for some $v \in \Sigma^*$ and $LEV(s,e) \leq t\}$ be the set of all entries represented by $\mathcal{T}$ that have a distance at most $t$ from the search word $s$.*

*Let $\mathcal{S} = \{e \in \Sigma^* \mid \langle e,v\rangle \in _GET(\rho,s,t,UNDEF, FALSE,FALSE,e)\}$ be the set containing all the projections of the first element of the pairs of the result list constructed by Algorithm 3.*

*Then, we have that $\mathcal{S} = \mathcal{E}$, i.e., that these two sets are identical.*

*Proof.* We split the proof into two parts: (i) soundness, i.e, $\mathcal{S} \subseteq \mathcal{E}$, and (ii) completeness, i.e., $\mathcal{E} \subseteq \mathcal{S}$.

For (i), for any string $e \in \mathcal{S}$, we need to show that (a) GET($\rho$,$e$) $= v$ for some $v \in Sigma^*$ and (b) LEV($s$,$e$) $\leq t$. Claim (a) can be proven straightforwardly by structural induction over the ternary search tree and Algorithm 2.

Claim (b) can be proven by induction over the threshold $t$. For the base case $t = 0$, Algorithm 3 is obviously equivalent to returning the result of Algorithm 2 as a singleton list.

For the step case $t > 0$, the induction hypothesis is that Claim (b) holds for $t-1$. The condition for the if statement marked as “edit allowed” in Algorithm 3 can be proven to evaluate to TRUE for any prefix $p$ of $s$ by straightforward structural induction over the ternary search tree and Algorithm 3. For a given prefix $p$, if an edit is possible at this stage, i.e., the remainder of $s$ without prefix $p$ is not the empty word for deletion and replacement, the three recursive calls marked as “insert”, “delete”, and “replace” are executed. In the case of “delete”, the induction hypothesis is immediately applicable. In the case of “insert” and “replace”, the induction hypothesis is applicable when the wildcard flag is consumed through the recursive call in the body of the if statement marked “consume letter”. Claim (b) thus follows from the observation that the prefix is concatenated with the strings from the resulting set in the body of the if statement marked “consume letter”.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

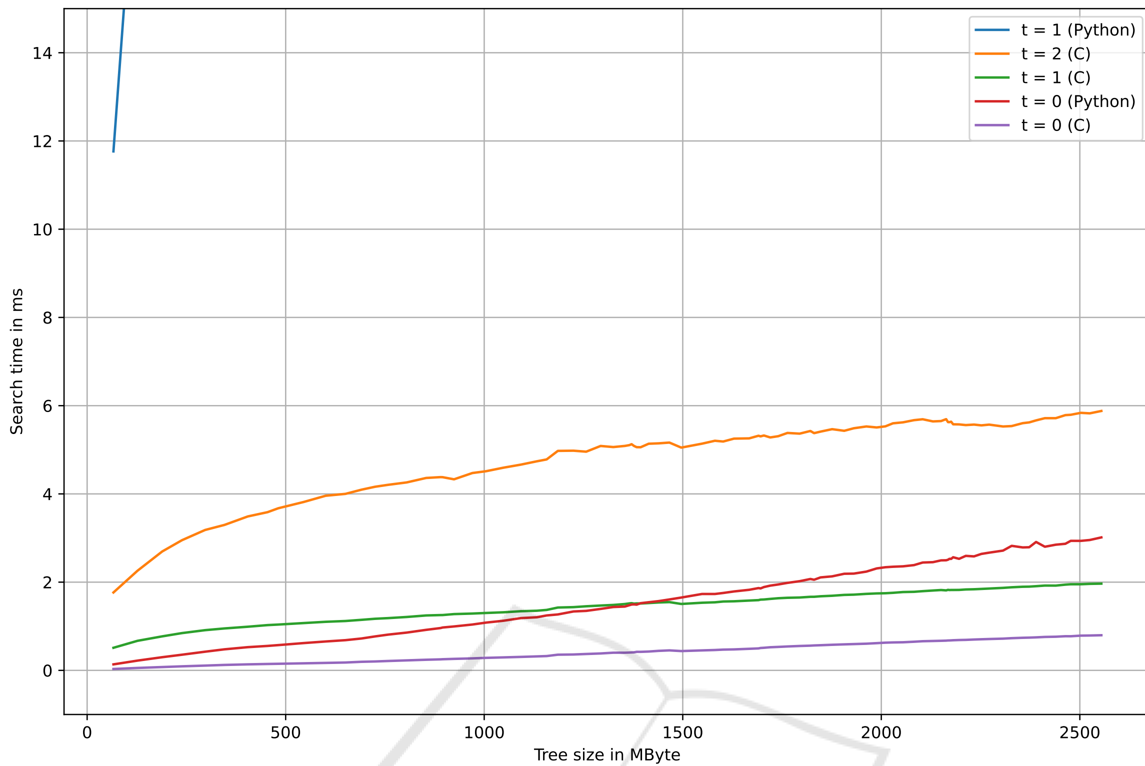



Figure 1: Mean search time depending on tree size for Levenshtein distances 0, 1, and 2.

For (ii), for any string $e \in \mathcal{E}$, we need to show that for some $v \in \Sigma^*$, we have $\langle e, v \rangle \in$ _GET$(\rho, s, t, UNDEF, FALSE, FALSE, e)$. Let $v$ be the result of executing Algorithm 2, i.e., $\langle e, v \rangle =$ GET$(\rho, e)$. The existence of $v$ follows from the definition of $\mathcal{E}$. The definition of $\mathcal{E}$ further provides that LEV$(s,e) \leq t$.

We proceed by induction over the threshold $t$ as for Claim (i) (b). For the base case $t = 0$, the claim follows again directly from the same observation as for the base case of the induction in Claim (i) (b). For the step case $t > 0$, the induction hypothesis is applicable for the same reasons as outlined in the step case of the induction in Claim (i) (b). □ □

Lemma 1 immediately implies the correctness of Algorithm 3.

**Theorem 1** (Correctness of Algorithm 3)**.** *Algorithm 3 is a correct implementation of approximate dictionary searching with Levenshtein distance.*

*Proof.* The GET procedure in Algorithm 3 calls the procedure _GET with the parameters ($\rho$,$s$,$t$,UNDEF,FALSE,FALSE). By Lemma 1 we obtain that GET returns all exact matches and matches with an edit distance of less than $t$. □

## 4 EVALUATION

An implementation of Algorithm 3 has been evaluated on the use case described in Section 1, i.e., on a dictionary with an alphabet of size 7040 and 162.193.908 entries. Instead of consuming at least 10 TByte with a trie datastructure, the ternary search tree for the full dictionary consumed just over 2.5 GByte. In initial experiments, the search algorithm with implicit Levenshtein automata (Algorithm 3) outperformed other memory-friendly data structures and algorithms such as Burkhard-Keller trees (Burkhard and Keller, 1973) regarding mean search time by at least an order of magnitude. For a real-world workload representing one month of user queries, the minimum query time was 0.004 ms, with a median of 0.077 ms, a mean of 2.146 ms, and a maximum of 53.858 ms. In addition to simulated workloads, real-world workloads, and stress testing, the performance was also evaluated qualitatively through the continuous involvement of end users and stakeholders (Sejr and Schneider-Kamp, 2021) at the company.

Figure 1 shows how the mean search time of Algorithm 3 depends on the the size of the ternary search tree, which was varied by performing experiments

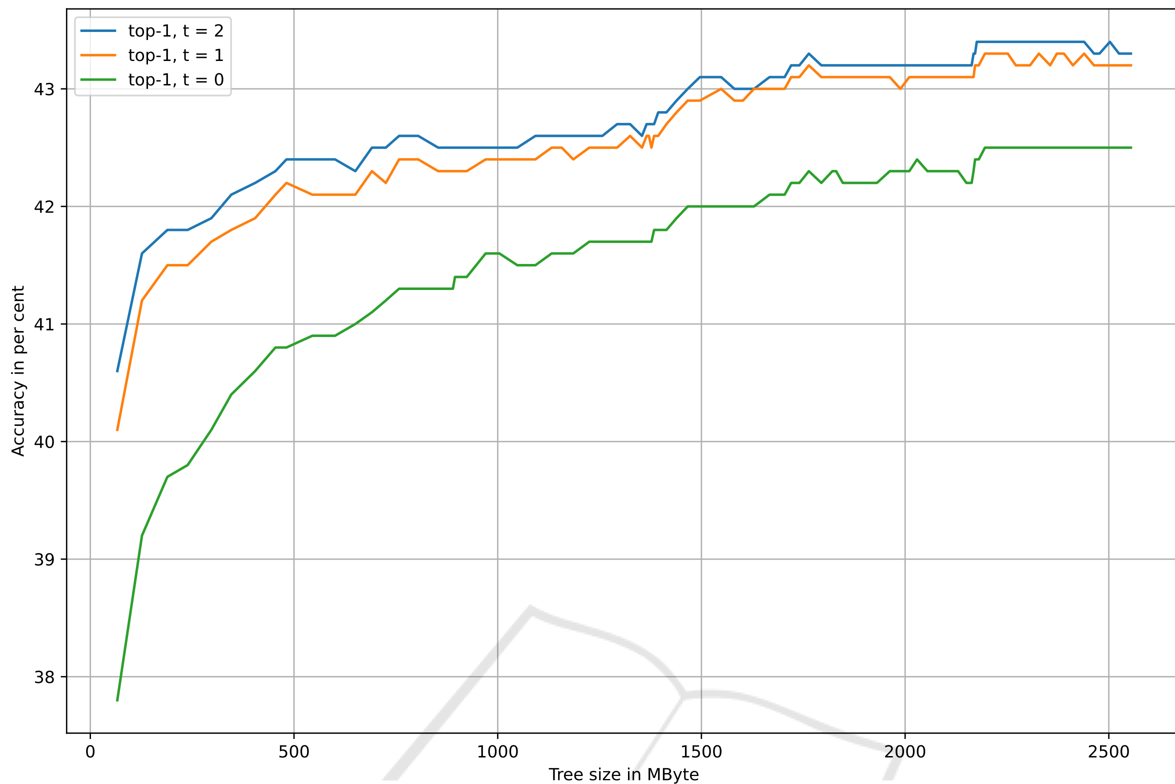



Figure 2: Accuracy depending on tree size for Levenshtein distances 0, 1, 2.

with 30 differently sized subsets of the full dictionary. A first implementation in the Python languag was found to be prohibitively slow for all but the exact case ($t = 0$). A second implementation in hand-optimized C performed an order of magnitude faster, facilitating the handling of thousands of simultaneous users on a single server instance with 4 GByte of RAM for a Levenshtein distance threshold $t = 1$ or $t = 2$.

The approximate dictionary searching in the considered application had the purpose of suggesting choices to users for misspelled search words. The top-$k$ accuracy for this application is defined as the proportion of cases in which the intended search word was among the first $k$ suggestions. Figure 2 shows how the top-1 accuracy depends on the size of the ternary search tree. The top-3 and top-10 accuracy reached approx. 62 and 76 per cent, respectively.

## 5 CONCLUSION

This paper has presented and proven correct a solution to the problem of approximate dictionary searching at a scale that relies on ternary search trees and implicit Levenshtein automata and scales well with the sizes of both alphabets and dictionaries both regarding runtime and memory usage.

## REFERENCES

Bentley, J. L. and Sedgewick, R. (1997). Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 360–369, USA. Society for Industrial and Applied Mathematics.

Boytsov, L. (2011). Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16:1.1:1.1–1.1:1.91.

Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

Leveling, J., Ganguly, D., Dandapat, S., and Jones, G. (2012). Approximate Sentence Retrieval for Scalable and Efficient Example-Based Machine Translation. In *Proceedings of COLING 2012*, pages 1571–1586, Mumbai, India. The COLING 2012 Organizing Committee.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88.

Schulz, K. U. and Mihov, S. (2002). Fast string correction with Levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85.

Sejr, J. H. and Schneider-Kamp, A. (2021). Explainable outlier detection: What, for Whom and Why? *Machine Learning with Applications*, 6:100172.

Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and Control*, 64(1):100–118.