Secure Systems Design and Technology Development. Veliko Tarnovo, Bulgaria
Proceedings of the International Scientific Conference, 2026. Volume III, 53-57

Improving Text Search Memory Efficiency with NSW Text Indexing

Dmytro Dashenkov
Software engineering department
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
https://orcid.org/0000-0001-9797-1863

Serhii Smelyakov
Software engineering department
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
https://orcid.org/0000-0002-5791-2479

Anastasiya Chupryna
Software engineering department
Kharkiv National University of Radioelectronics
Kharkiv, Ukraine
https://orcid.org/0000-0003-0394-9900

Loreta Savulioniene
Faculty of Economics
Vilniaus kolegija
Higher Education Institution
Vilnius, Lithuania
l.savulioniene@viko.lt

Dainius Savulionis
Faculty of Electronics and Informatics
Vilniaus kolegija
Higher Education Institution
Vilnius, Lithuania
d.savulionis@eif.viko.lt

Paulius Sakalys
Faculty of Electronics and Informatics
Vilniaus kolegija
Higher Education Institution
Vilnius, Lithuania
p.sakalys@eif.viko.lt

***Abstract*— Modern day full text search relies on a simple yet robust inverted index data structure to find documents matching a text query. However, this approach is limited by the required spelling checker, which often has memory requirements comparable to the index itself. In this work, a novel approach to text indexing is introduced. Based on the navigable small world data structure, it offers a two-in-one solution for search with built-in spelling correction, sacrificing some performance for a substantial memory economy. The proposed index, just like the conventional inverted index, uses separate tokens extracted from a document as the main building blocks. The tokens are organized in a graph, each node representing one token. The nodes list all the document identifiers for the documents where the token is found. The edges of the graph connect tokens that are similar in terms of edit distance. The proposed approach utilizes an arbitrary ranking function to find relevant documents, the state-of-the-art BM25 ranking function is recommended as a reasonable default. The function values are weighed to take the edit distance into account. The ability to process misspelled words allows avoiding maintaining a separate structure for spelling correction while still generating search results which account for possible misspellings in the query phrase. The search time losses are attributed to the neighbor exploration while traversing the navigable small world data structure, and thus can be mitigated by configuring the data structure to maintain a fairly low number of connections. Such a configuration also provides a hard limit to the number of edits allowed in a single query term. The paper discusses in detail the ways of picking the optimal parameters for the algorithm, as well as the limitations of the proposed approach.**

Keywords*— *full text search, memory efficiency, navigable small world, spelling correction.

I. Introduction

A. Motivation

Full text search is a computer science task of finding relevant documents is a large text corpus based on an arbitrary subset of words. With the modern developments if machine learning which allow high accuracy semantic search, based on embedding vector closeness, rather than word matching, full text search can still be used to increase search accuracy using a hybrid approach, where both semantic and full text results are gathered and ranked as a part of one set of search hits. In this paper, full text search is considered as a separate procedure, detached of the possible hybrid approaches.

Usually, text search includes these high-level steps:

1. Query spell checking and correction.
2. Stop word exclusion.
3. Query tokenization (stemming).
4. Inverted index lookup.
5. Hit ranking with a function, such as BM25.

Online ISSN 3044-7771
https://doi.org/10.68302/std2026.vol3.215

The described algorithm can be adjusted based on the needs of a specific system, for instance, a filter matching can be added typically after the inverted index lookup to skip certain documents based on a business logic rule.

The inverted index is a simple yet powerful data structure, which consists of a hashmap of tokens to sets of documents, most often, represented by document IDs, which contain the associated token. The structure allows maintaining large text corpus indexes with constant lookup time and constant in the best case and linear in the worst case insert time. Additional optimizations such as bloom filters and roaring bitmaps can be used to further speed up the lookup time.

However, the inverted index can only work for well-prepared text data on both the corpus and query sides. When dealing with user-generated data, spell checking is a key step of ensuring that full text search will yield expected results, effectively enabling any full text search setup to be fuzzy. Modern approaches to spell checking, such as the symmetric delete spelling correction algorithm, can achieve great performance yet leave a sizable memory footprint due to specialized data structures, which they build to operate efficiently.

During the past months, at the time of writing this paper, rapid access memory hardware is going through a phase of severe price increases. In this light, it becomes paramount to develop and implement into real-world production systems, approaches to various computer science problems, including full text search, which would reduce the need for memory intensive computations. In this paper, we introduce an approach to full-text search, that would severely decrease memory consumption for a price of a small lookup speed loss.

B. Related Works

Some of the remarkable works in the field of full text search improvements include the work [1], which uses pre-processing techniques on inverted indexes in order to make the full text search robust to typos, achieving quite low latency figures for the search procedure.

Another work [2] introduces the concept of interactive fuzzy search, which utilizes prefix-based lookup with error-correction, allowing the system to execute search while the user is typing.

Another approach to the task is evaluated in [3], which tests spelling correction approaches as a part of the ranking phase of the search, rather than the retrieval phase. This paper suggests that a standalone spelling correction preceding the search operation could, in some cases, hurt the retrieval accuracy.

Despite this, [4] supports the notion of spelling correction being paramount to the search process, indicating that most improvements to modern full text search lie in the usage of an appropriate spelling correction mechanism.

An alternative approach to index memory efficiency is introduced in [5], leveraging complex compression mechanisms in order to reduce the inverted index memory footprint. As the approach proposed in this article does not get rid of the inverted index entirely, the methods described in [5] could prove useful for practitioners implementing the approach.

Works [6] and [7] illustrate the versatility of vector-based text indexing, highlighting the effectiveness of interoperability of semantic and text-based approach to processing large corpora.

Finally, recent developments in intellectual data analysis, laid out in [8], suggest that processing large amounts of data in a relational fashion, which, could also be presented in graph form, proves useful for storing and indexing purposes.

II. Materials and Methods

A. Proposed solution

The proposed approach uses a navigable small world (NSW) data structure to index text data. NSW is an undirected graph which stores data items in nodes. The edges connect nodes which are closest to one another by definition of a selected closeness function. In the case of full text search, each node contains a word, or, more precisely, a token and the links the documents where the token occurs. The edges connect tokens which are closest to one another by edit distance.

Typically, in a NSW graph, there is an absolute limit count of edges one node may have. However, in the proposed approach, this parameter is replaced by the more meaningful max edit distance parameter, which limits how many edits could be done to token for the edge connection to still be required. The recommendation on selecting this parameter are provided in the Experiment section of this paper. Other parameters that are typically attributed to the NSW data structure are not relevant in this case.

Since each edge of the graph represents one or more edit operations, the edges are weighted with the edit count. This is an important property which allows using a ranking function which accounts for spelling correction. If such a function is not needed, the edge weights could be dropped.

It's worth noting that the graph could, depending on the text corpus and the chosen max edit distance, turn out to be disconnected. This is, however, not significant for practical purposes, since all the lookup operations will be limited to local exploration, as described below.

In order to access each specific token in the graph, a hashmap of string tokens to nodes is maintained.

The algorithm for building the NSW graph involves precalculating the edits that could be possibly made to each given token. The steps of the algorithm are as follows.

1. A new document is introduced. Tokenize the document with possible stemming in order to receive a set of district normalized tokens.
2. For each token, check if it is already present in the graph by querying the hashmap. If it is, add the reference to the new document (in practice, the document is referenced by a unique ID) to the node and continue to the next token.

3. If the token is not yet present in the graph, create a new node for the token and add a reference to the document.
4. Iterate over all the possible variants of the token, obtained by making the edit operations to the token's string value until the selected max edit distance is reached. Check if each of the variants is present is the graph. For the found nodes, create an edge between the found node and the new node.
5. Prune the repetitive edges by traversing the graph starting with the new node and deleting edges with a higher weight. This way, only one optimal path between two nodes is maintained at any time.

The need to create edges which represent two or more edit operations stems from the fact that some tokens do not have a valid path between them with only one edit hops. Consider words "cat" and "rat", which can be connected with a single edge with the weight of one, the words "cat" and "rate", which are connected with two hops, "cat" to "rat", and "rat" to "rate", and the words "cat" and "came", which can only be connected with a single edge with the weight of two, since neither "cate" nor "cam" are words that occur in a given corpus.

Due to the need to explore all the possible variants of the token within the edit distance, the time complexity of the insertion of a document is calculated as the average token size to the power of max edit distance and then multiplied by the document size, as shown in formula (1).

$$T_1 = O(m\ n^d), \tag{1}$$

where T_1 is the time complexity, m is the size of the document, n is the average size of the token, and d is the chosen max edit distance.

On practice, iterating over all possible edits is very time and memory consuming task. In order to optimize it, the corpus is first fed symmetric delete spelling correction algorithm, also known as SymSpell, which builds a special data structure for the edits. Then, while filling the graph, the SymSpell structure is used to locate all the known valid edits, significantly reducing the time needed for the index construction. After the index is built, the SymSpell structure is removed from memory.

Another practical time and memory optimization lies in scaling the effective edit distance for each token individually, depending on the token length. This means that the max edit distance becomes an upper limit for the effective edit distance, and 1 is taken and the lower limit, and the effective edit distance is computed using formula 2:

$$d' = min(d, max(1, s / 3)), \tag{2}$$

where d' is the effective edit distance, d is the configured max edit distance, s is the length of the token.

The search algorithm, however, utilizes the prebuilt NSW graph to jointly execute both the lookup and the spell checking, bringing in better results for accidentally misspelled words. Its worth noting that the algorithm does not handle words that are misspelled to the point of not being valid tokens, but rather handles the case when the intended word is spelled as a similar yet different word. The Discussions portion of this paper contains an in-depth explanation of why this approach is useful. The search algorithm consists of the following steps.

1. Stop word exclusion.
2. Query tokenization (stemming).
3. For each of the tokens, the corresponding NSW graph node is looked up using the hashmap.
4. All the documents referenced in the node are added to the results to be ranked.
5. All the edges connecting to this node are explored. The graph is traversed in order of edit distance from the original node, from smallest to highest. All the found documents are added to the result for ranking with their edit distances memoized.
6. Once all the nodes are explored, the documents are ranked, and the rank values are weighed to include the information about the edit distance.

The lookup operation has the worst-case scenario time complexity of average query token size to the power of max edit distance then multiplied by the query size, as shown in formula (3).

$$T_2 = O(q\ n^d), \tag{3}$$

where T_2 is the time complexity, q is the size of the query, n is the average size of the token, and d is the chosen max edit distance.

In the best-case scenario, however, the node would not be connected to any other nodes and thus the time complexity would be linear, depending on the query size only. This best-case scenario, however, means that the NSW data structure did not bring any value to the search, since no misspelled variants were considered.

When ranking the collected hits, a BM25 function should suite most cases due to its versatility and high variability [9]. The computed rank scores for each token are multiplied by the inverse edit distance from the query token to the token from which each hit is resolved, as shown in formula (4).

$$f = \Sigma^N_{i=0}\ (BM25(t_i) / (d_i + 1)), \tag{4}$$

where f is the rank score of the hit, N is the number of query tokens, $BM25(t_i)$ is the rank score of the hit with regards to a specific token, d_i is the edit distance between the original query token and the token found in the document.

B. Experiment design

In order to check the suggested approach experimentally, a comparison is drawn between two search indexes.

Index A is built using the conventional method. Firstly, the corpus is split into separate words and stemming each word. An inverted index is built, maintaining a mapping between the resulting tokens of the corpus and the documents that contain those tokens. Next, a spelling correction data structure is built for the symmetric delete spelling correction algorithm. In this and all other occurrences, in order to make sure the spelling code is robust and as close as possible to a real-world scenario, the SymSpell component of the LanguageTool library [10] is used to build such a data structure and generate spelling corrections. A search operation on index A would involve tokenizing the query, looking up possible alternative spellings for each query token, gathering relevant documents via the inverted index, and finally ranking the documents.

Index B is build using the approach suggested in this paper. Firstly, just like in the index A process, the corpus is tokenized by splitting the texts into words and stemming each word. Next, an NSW graph is built with the tokens, as described above. In order to speed up the graph building process, a temporary SymSpell instance is created with all the tokens of the corpus. This is a shorthand way of generating valid alternative spellings for each token, once the NSW graph is built, the SymSpell instance is deleted, freeing up memory.

The corpus chosen for the test is the corpus of Simple English Wikipedia articles. A 10K article subset is used, containing 42 unique tokens and 42 total, averaging 42 tokens per article. Each article consists just of the title and a plain text content, both indexed equally for this experiment.

III. Results and Discussion

A. Experimental findings

The proposed approach has one important parameter, the edit distance between the tokens. In order to compare the memory usages of the conventional and the proposed approaches, a series of experiments with different values of max edit distance was conducted. The results of the described experiments are provided in the table 1.

TABLE 1 INDEX MEMORY FOOTPRINT

Max edit distance	Memory Footprint, MB	
	Index A	***Index B***
0 (no spell checking)	22	25
1	22	45
2	22	52
3	108	62
4	110	65
5	118	66

As shown in the table, across all the edit distance limits, the NSW-based approach (index B) requires less memory to store all the required indexes that the conventional approach (index A). The main reason for this is the fact that the SymSpell data structure requires more information to error-correct queries than the NSW graph, since the graph only stores valid misspellings, while the SymSpell stores all the possible edits of a token.

Next, the search speed was measured. Table 2 provides latency per 100 search operations with the same query phrases for all the configurations described above.

TABLE 2 SEARCH LATENCY

Max edit distance	Memory Footprint, MB	
	Index A	***Index B***
0 (no spell checking)	6	54
1	15	73
2	17	166
3	23	1318
4	59	2066
5	185	2476

The figures provided in table 2 are merely indicative of the amount of work required for each search configuration. The measurements were conducted in single-thread environment with no attempts of optimization, yet they allow understanding the scale of the trade-off when switching from a more conventional indexing approach to the suggested one. Time for scoring hits with a ranking function, such as BM25 is not included, as it is not variable between the index methods.

B. Discussions

In this paper, the edges of the graph represent edit operations. Edge weights are always integer and represent the edit distance between the two connected word nodes. However, in practice, there might be some value in distinguishing different text edits and assigning different weights to them, based on, for instance, probability of a typo that would cause that edit. In this scenario, a missing letter would have one probability, an extra letter may have another probability, and the letter swaps would each have their own probabilities, based on how often do people mistype one word or another. This might provide additional potency to the spell-checking aspect of the algorithm, allowing different typos to influence the final ranking score more or less depending on their likelihood. Further research is needed to compile a dataset of typo probabilities and test this approach.

This paper describes spell checking a search algorithm with integrated spell checking. However, the spell-checking aspect is limited to switching between valid tokens and does not cover the case of completely misspelled words. In the modern practical applications, this is often enough, since queries might go through a client-side spell checker, which would eliminate any misspelled words but retain words that are present in the spell

checker's dictionary, allowing typos like "cat" – "rat" to happen.

If the assumption of a client-side spell checker cannot be made, a simple modification is suggested to the graph building algorithm, which allow the index to support any kind of misspellings. For this, is an edited variant of the token is not found in the graph, it must be created and linked to the original token. No documents are linked to that extra node, signifying that it does not represent a valid token, but is merely a synthetic node. Now, if a misspelled query token is passed to the search, the graph traversal will begin with this synthetic token and the search will proceed from it to the tokens with linked documents. Given this modification to the data structure, if a query token is present in the index, it is guaranteed that some result hits will be passed to ranking. The opposite is true as well, when a token is not found in the index, no hits will be produced for that token. This is a good baseline for most systems, though, it comes at a cost of using extra memory for storing the synthetic nodes.

All things considered, the suggested approach offers an adequate alternative to modern methods of indexing text data. Another benefit of constructing an NSW graph is the interoperability with the hierarchical navigable small world (HNSW) graph indexes built for semantic search. Since the two data structures have similar properties (HNSW being a more advanced version of NSW), it is easy to consider a hybrid approach, where one graph hosts both document vectors and separate tokens, allowing to search via both modes in a single traversal operation. In this case, hierarchical nature of the HNSW graph would provide an obvious entry point to the traversal, potentially eliminating the need for both nodes with no linked documents and a hashmap pointing to all each of the nodes. Given the rise of popularity of hybrid approaches to search, the coexistence of the two indexes in one structure could prove useful in many practical applications. The details of such coexistence must be further researched, however, similar approaches are suggested by researchers behind the Allan-Poe graph index, which combines text, vector, and knowledge graph information all in one data structure [11].

IV. Conclusions

In this paper, a novel approach to indexing text data is proposed, which spelling correction with search. The approach shows a more efficient memory usage for the static data structures (indexes) when compared to a more conventional correct-then-search approach. Also, by connecting tokens that are close by edit distance to one another, the user of the index gains the ability to correct typos, allowing for a more precise search, when a query token which contains a typo is accompanied by one of a few tokens which do not. Thus, the approach provides both memory savings and formalizes fuzzy ranking formula using BM25 weighed with respect to the edit distance between the query token and the hit token.

The suggested approach compromises on search speed when compared to non-spell-checking search, but matches the search of a spell-checking approach with similar spell-checking mechanisms.

The promising notion of combining the NSW approach with a vector-based semantic search would further improve state-of-the-art hybrid search systems, consolidating the two search modes into one graph traversal operation, which, in order, simplifies ranking. Such approach is already explored by other researchers and could yet still be improved with an edit-distance-based edge building strategy.

References

[1] H. Bast and M. Celikik, "Efficient fuzzy search in large text collections," ACM Transactions on Information Systems, vol. 31, no. 2, pp. 1–59, May 2013, doi: https://doi.org/10.1145/2457465.2457470.

[2] S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," The Web Conference, Apr. 2009, doi: https://doi.org/10.1145/1526709.1526760.

[3] I. Zelch, G. Lahmann, and M. Hagen, "Embedding-based Query Spelling Correction," in WOWS'24: 1st International Workshop on Open Web Search, Glasgow, Scotland, Mar. 2024. Accessed: Apr. 08, 2026. [Online]. Available: https://ceur-ws.org/Vol-3689/WOWS_2024_paper_4.pdf

[4] S. K. Singh, "Spelling Correction in Healthcare Query-Answer Systems: Methods, Retrieval Impact, and Empirical Evaluation," arXiv.org, Feb. 2026, doi: https://doi.org/10.48550/arXiv.2603.19249.

[5] C. Kamphuis, P. de Vries, L. Boytsov, and J. Lin, "Which BM25 Do You Mean? A Large-Scale Reproducibility Study of Scoring Variants," in Advances in Information Retrieval, Springer Science+Business Media, Apr. 2020, pp. 28–34. doi: https://doi.org/10.1007/978-3-030-45442-5_4.

[6] R. Lynnyk, V. Vysotska, Z. Hu, D. Uhryn, L. Diachenko, and K. Smelyakov, "Information Technology for Modelling Social Trends in Telegram Using E5 Vectors and Hybrid Cluster Analysis," International Journal of Information Technology and Computer Science, vol. 17, no. 4, pp. 80–119, Aug. 2025, doi: https://doi.org/10.5815/ijitcs.2025.04.07.

[7] V. Vysotska, K. Smelyakov, A. Chupryna, M. Derenskyi, V. Repikhov, and M. Hvozdiev, "AI assistant for intelligent interaction and route optimization in offshore turbine maintenance system," Proceedings of the PhD Workshop on Artificial Intelligence in Computer Science at 9th International Conference on Computational Linguistics and Intelligent Systems (CoLInS-2025), vol. 4015, May 2025, doi: https://doi.org/10.31110/colins/2025-3/001.

[8] V. Filatov, O. Zolotukhin, and M. Kudryavtseva, "Intellectual data analysis in relational information and analytical systems," Innovative Technologies And Scientific Solutions For Industries, no. 4(34), pp. 101–111, Dec. 2025, doi: https://doi.org/10.30837/2522-9818.2025.4.101.

[9] G. E. Pibiri and R. Venturini, "Techniques for Inverted Index Compression," ACM Computing Surveys, vol. 53, no. 6, pp. 1–36, Feb. 2021, doi: https://doi.org/10.1145/3415148.

[10] "languagetool-org/languagetool," 2024, gitHub repository. [Online]. Available: https://github.com/languagetool-org/languagetool

[11] Z. Li, Y. Li, Y. Zhu, C. Ge, Z. Chen, and Y. Gao, "All-in-one Graph-based Indexing for Hybrid Search on GPUs," arXiv.org, 2025, doi: https://doi.org/10.48550/arXiv.2511.00855.