
Production Systems and Information Engineering
Volume 9 (2020), pp. 5-18

EFFICIENCY ANALYSIS OF NNS METHODS

Balázs Bolyki
University of Miskolc, Hungary
Student of Computer Science Engineering
bolyki@iit.uni-miskolc.hu

Dávid Polonkai
University of Miskolc, Hungary
Student of Computer Science Engineering
polonkai3@iit.uni-miskolc.hu

Dr. László Kovács
University of Miskolc, Hungary
Department of Information Technology
kovacs@iit.uni-miskolc.hu

[Received ... and accepted ...]

Abstract. Nearest Neighbor Search is a key operation in multiple information technologies fields, for example, string matching, plagiarism detection, natural language processing, image clustering, etc. It is crucial, that we have cost efficient methods and structures for retrieving data based on similarity. We conducted a survey of two popular – Vantage Point tree and Locality Sensitive hashing –, and one more recent – Prefix tree with clustering – NNS methods. In order to perform a wide range of tests on these algorithms, we adapted each to Python language, and developed multiple tests. In this paper we present the description of the three algorithms and the results of our tests. We aim to provide an informative comparison of the three major Nearest Neighbor Search structures.

Keywords: NNS, VP-tree, LSH, Prefix tree, Nearest Neighbor, search, comparison

1. Introduction

The search for similar objects is a key operation in general information management. When we retrieve information based on the high level of similarity – or in other words, the smallest distance – between known and unknown data,

we execute Nearest Neighbor Search (NNS). The applications of NNS are numerous and also are the methods invented to make it faster, more effective. The query for nearest neighbor elements is used, among others in autocomplete and spell checking [1], plagiarism detection [2, 3, 4, 5, 6], natural language processing [7], image clustering [8], and medical databases [9]. The time efficiency of NNS operations is a crucial cost factor in the whole information system. Similarity and distance are two measures approaching the same problem from the opposite directions. The more similar two objects are to each other, the less the distance is between them, while the greater the distance between them, the less similar they are. Therefore we use both measures in this essay.
This work focuses on an important application area, the search in a word repository. This paper's main purpose is twofold, first to analyze the cost efficiency of the known NNS methods and to adapt them to the investigated problem domain, namely the similarity search in word repositories. The work analyzes two popular methods, VP- tree and LSH algorithms and compares them with a recent approach, the prefix-tree NNS structure. The performed tests cover both time efficiency and search accuracy analyses. Our goal is to list, explain and evaluate these methods and to give the readers an insight into the strengths and weaknesses of each.

Nearest Neighbor Search. Nearest Neighbor Search (NNS) is the operation, which retrieves a data object, from a given dataset, based on its distance from a query object. Formally, if $\mathbb{U}$ is the universe and $\mathbb{S} \subseteq \mathbb{U}$ is the dataset, in which we search, and $q \in \mathbb{U}$ is the query object, we can define NNS as follows.

$$d : \mathbb{U}^2 \to \{x \in \mathbb{R} \mid x \geq 0\} \tag{1.1}$$

$$\begin{aligned} N_1(q) = \{y \in \mathbb{S} \mid \forall z \in \mathbb{S}, d(q,y) \leq d(q,z)\} \\ |N_1(q)| = 1 \end{aligned} \tag{1.2}$$

$$\begin{aligned} N_k(q) = \{\forall x \in N_k(q), y \in \mathbb{S} : d(q,y) < d(q,x) \Rightarrow y \in N_k(q)\} \\ \text{and } |N_k(q)| = k \\ \text{where } k \in \mathbb{N}^+ \text{ and } k \leq |\mathbb{S}| \end{aligned} \tag{1.3}$$

Formula 1.1 defines a distance function d, which we use in the Nearest Neighbor Search definition in the Formula 1.2. In Formula 1.3 we also define the generalization of NNS to k-NNS, where we retrieve the k nearest neigbors of the query q, from the dataset $\mathbb{S}$.

2. Description of evaluated algorithms

The three evaluated approaches, Vantage Point tree (VP-tree), Locality Sensitive Hashing, and Prefix tree with clustering (prefix tree) each realize an efficient way to execute nearest neighbor search operation, however, their

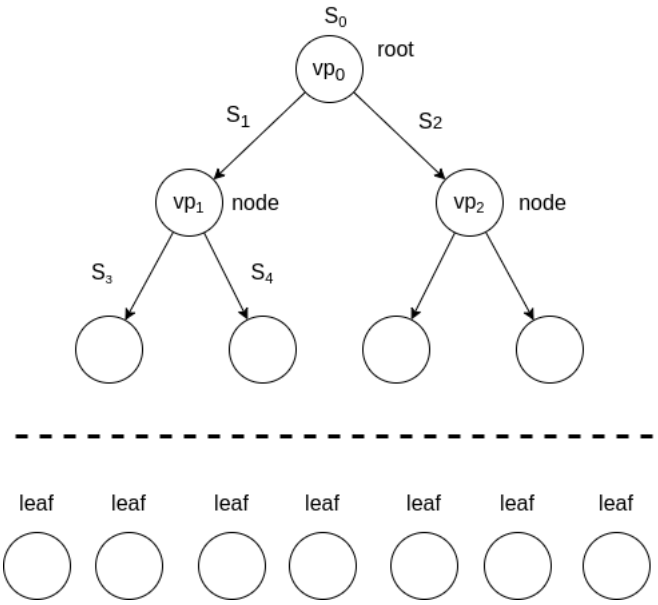

Figure 1. Representation of VP-tree concept

approach is quite different. In this section we will describe the approach of each algorithm generally.

2.1. Vantage Point tree

VP-tree realizes NNS operation based on the concept of general metric spaces. It requires the dataset containing the data objects and the distance function d to build the indexing structure. Here, the distance function must satisfy the properties of a metric (positivity, identity of the same objects, symmetry, and triangle inequality), because the VP-tree can use the triangle inequality to prune branches, when searching.
Our main reference for the VP-tree is the paper [10]. The main concept of the VP-tree algorithm to solve the NNS problem is to build an indexing structure in such a way that the data objects get distributed according to their distance from vantage points, which are chosen from the dataset, one at each non-leaf node. At the building stage if the current dataset S_i does not fit into a leaf, we apply Formula 2.1, where we denoted the base dataset as S_0 and the created subsets are S_1, S_2 and S_i in further nodes, while m is the median of the distances from the vantage point (the distance between the vantage point and the elements of S_i is calculated for each element during the partitioning at each node). This partitioning scheme would be recursively repeated at each

node, until all created subsets are allocated into leaves.

$$\begin{aligned} S_1 &= \{s \in S_0 \mid d(s, vp_0) < m\} \\ S_2 &= \{s \in S_0 \mid d(s, vp_0) \geq m\} \end{aligned} \tag{2.1}$$

The algorithm for searching in the tree is presented in the paper [10]. It uses a recursive strategy to traverse the tree and prunes branches of it based on the triangle inequality.

2.2. Locality Sensitive Hashing

The Locality Sensitive Hashing is based on hash functions. The hash functions map a key into a hash value. The hash function represents the same value for the same input. It is possible, that two different keys mapped into the same hash value. This is hash collision. [11]
The LSH uses hash collision to find similar keys. This method hashes data objects by multiple hash functions and stores the hash value and key pairs. Therefore if more collisions are found in different hash functions, it is more probable that keys are similar. To find similarities the algorithm hashes the query point and returns the elements from the buckets that contain that point. [12] This method also uses minhash. It is a special hash function, that we execute multiple times to receive the same hash values for similar keys. In our case it is used to map the keys into hash values, which we add to the LSH function. For better understanding we have to declare shingling, which is a method, that can divide its input into k lenght sequences. After shingling the hash value, the method puts them into the buckets of the LSH function. The idea of the LSH algorithm is represented in Figure 2. In our case the algorithm

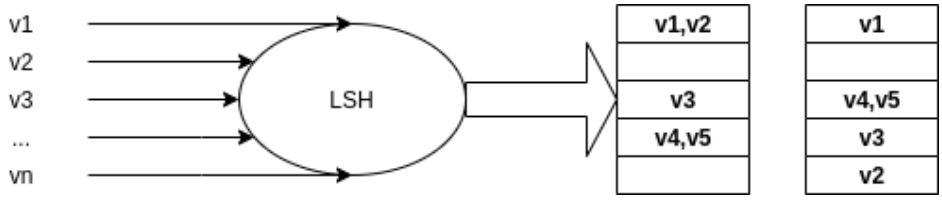

Figure 2. Representation of Locality Sensitive Hashing concept

we use is a library based on the [13].

2.3. Prefix-tree with clustering

This method uses clustering for preprocessing and the prefix-tree for storing the actual words. The prefix-tree is a special kind of tree, in which the nodes are letters. A word is represented by the path of letters.

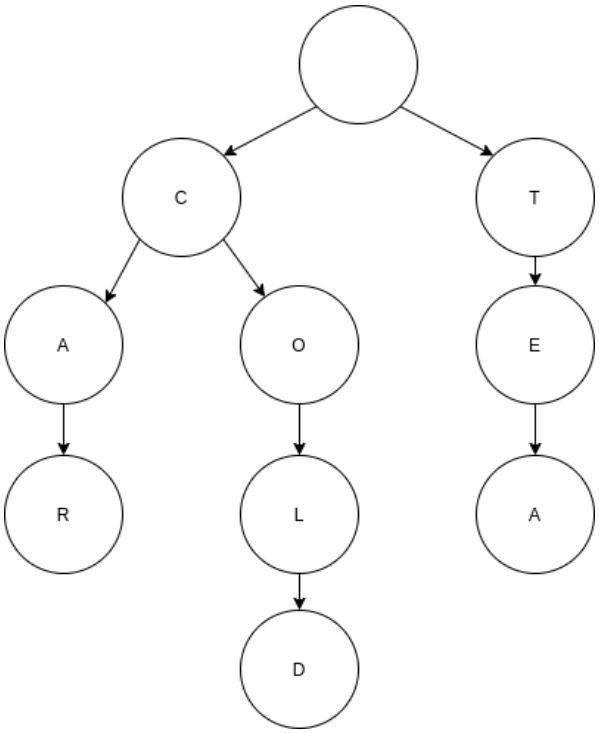

Figure 3. Representation of Prefix-tree concept

Prefix-tree. In case of insertion the method starts to go down in each branch and finds the current letter in the structure. If there is no such a letter, it creates a new node. So for example if we want to find the *word*, then in the first level from the root we look for a letter *w*. In the query for 1NN search we go down in each branch and check if the current character is the same in the structure. If they are different, it means that the distance between them is one more. By this method we can find the best fitting similarity in the structure, but if we go down in every branch it costs much in time, so the method also uses a limit parameter, which determines the maximum distance in a branch. If the distance reaches this value, we cut the branch.

Prefix-tree with clustering. This method also uses clustering. This means that we map the words into different sets. These sets are further from each

other and does not overlap. In each cluster there is a tree and the words are stored there. The stucture building starts with calculating the central points of the clusters. After this we build all the prefix trees. In case of insertion the algorithm calculates the nearest cluster and builds the word into the tree. For calculation the method represent the words into vectors and runs a minimum search on the cluster distance from the word in order to find the nearest cluster. The query method works the same, so it finds the cluster, then goes down in the tree and searches for the word as described above. The base implementation of the method is presented in [14].

3. Adaptation of investigated methods

In order to test the methods, we adapted them to a common implementation framework. We chose the Python programming language, because it provides libraries and partial implementation of the used methods, and it hides most of the lower level operations, so we could focus on the unique parts of the algorithms.
The parameters of the computer in which the tests have run are the following:

- cpu: Intel®Core™i7-8550U @ 1.80GHz
- ram: 7.7 GiB @2400 MHz and 7.5 GiB of swap memory
- swap device: M.2 Solid state drive with 600 MB/s data transfer rate
- operating system: Ubuntu 18.04.3 LTS 64-bit

3.1. Vantage Point tree

Our VP-tree implementation is based on the paper [10]. We should also reference [15] for the implementation of the Autosorting list and as a structural example. However, due to our different platforms and emphasis – namely us using mainly Python language and our tests focusing on speed rather than page access rates – we diverged from [10] in multiple aspects. We followed the proposition of the paper for

- building the tree and
- executing k-NNS in the tree.

We performed the following extensions of the base model:

- We set the leaf size based on the quantity and not the size of the data objects, that would be contained within. This is more advantagous in a higher level programming language, since it provides more information, than does the data object size.
- We designed our own insertion algorithm. This looks up the leaf, in which the data object to be inserted should be placed, and allocates it

in the leaf, it still has space. If does not have space, then the algorithm starts to go up the tree, and checks if the subtree marked my the checked node (its descendants are its subtree) has space. If it has, the algorithm retrieves the data stored in that subtree, then rebuilds it. If the entire tree is full, then the entire tree is rebuilt.

3.2. Locality Sensitive Hashing

The algorithm originally used for similarity search in big texts like paragraphs. So we have to tweak the parameters to work with word similarity search. One of the most important parameters is the `permutations`, which determines the number of permutations by the minhash function. If this parameter is big, then the memory consumpsion and building time of this method increase, but the query time will be less. We determined that the ideal number for this parameter is 60. The other parameter is `n_gram`, which determines the shingle size for minhash. We choose this to be 2, because by this we can enter bigger than 2 lenght words. To make sure that the words contain more than 2 characters we built a function, which fills the words, which are less than 4 characters with _ characters. Later in the query we also run this on the searched word, to make sure that the right results are returned. We also changed the `no_of_bands` parameter, which defines the number to break the minhash signature before hashing into the buckets. The accuracy is affected by this parameter. We set the sensitivity to 2 which means the number of buckets texts must share to be declared as similar. Moreover we implemented the k-NNS methods, which calculates the k nearest from the result set.

3.3. Prefix-tree with clustering

In case of this method we also modified the parameters. The most important parameter for this method is the `limit`, which determines the maximum distance that we want to search in the tree. As is mentioned this parameter determines how deep the search can go down to find the best match in a branch. The parameter has to be larger than the distance of the searched word and its nearest neighbour, otherwise the algorithm does not return a word. But if the `limit` is much larger than the nearest neighbor, then the search time increases dramatically. In most of our tests we declared the limit parameter as 3, in some of them we used also 3 and 6. The other crutial parameter is the number of clusters. Because the word cluster determination is a step with high costs we have to choose it properly. our wordlists use 50000 to 2.4 million words, therefore we choose a rather big clusternumber, which is 100. In addition to the parameter settings we created methods for k-NNS

search and for the insertion. The k-NNS search method goes through the tree until k number of results are found or the limit distance reached.

4. Performed tests

To test the algorithms on similar word searching, we have to create wordlists. These wordlists contain Hungarian and English words. The building wordlist consists of 2.4 million Hungarian words. In order to build structures with less words we take a sublist from this. The searchlist is a Hungarian and mostly English wordlist. These words are not in the building list.
We created several tests:

- build time tests for different wordlists,
- search time tests for different wordlists and different percentage of known-unknown words,
- k-NNS tests: scaling depending on the k value;
- accuracy tests,
- insert time tests.

Most of the tests measure time required to do the task, except for accuracy tests, which measure that the returned word is the real nearest neighbour.

Build test. The building time test is, where we tested the required time to build up the structure from zero. We ran this test on different wordlists, from 200000 to 2.4 million in each cycle we increased the number of words by 200000. The results of the test is represented in Figure 4.

Accuracy comparison. We measured the accuracy of each algorithm by first building a structure from a wordlist of 50000 words, then executing nearest neighbor search in two ways. We retrieved the nearest neighbor using the evaluated algorithm, and also using a linear search algorithm, that we wrote for this purpose. We accepted the result returned by the linear search algorithm as the real nearest neighbor, and compared it to the result returned by the evaluated structure. If the two returned objects were of the same distance from the query point, then we considered the search accurate, otherwise we considered it inaccurate. We looked up the nearest neighbor of 20 words, and calculated the percentage of accuracy. We also executed this test for different proportions of known-unknown words in the wordlist (for example, 20% of the search list was also in the structre). The results are presented in Figure 5.

Search test. The search time tests measure the time required to run the query method. We used 20 words size wordlists, and measured the time each algorithm takes to look up the nearest neighbor of the 20 words, then divided

the result by 20 to receive the average search time per word value for each algorithm. The results for the 20 unknown words (words not inside the structure) are represented in Figure 6.

K-NNS test. We tested the algorithms for their scaling given multiple k values, when we execute k-NNS operation. To do so, we built the structures from 50000, 100000, ..., 400000 words size wordlists, then executed k-NNS, with the above mentioned 20 words search list (none of the 20 words were inside the structure), with the k values of 1, 3, 10. As before, we present the average search time per query word in Figure 7.

Insert time test. The insert time test is a test, in which we measure the time needed to insert 100 words into the different structures. We executed this insertion on structures built from 200000 to 1.2 million words size wordlists. The results of the test are represented in Figure 8.

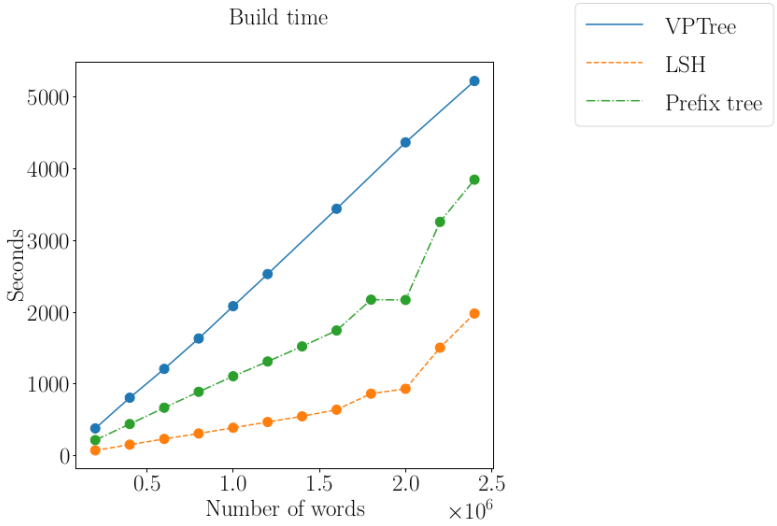

Figure 4. Building time comparison

5. Conclusion

In this article we have listed a few application areas of Nearest Neighbor Search, described a few import Nearest Neighbor Search algorithms, namely:

- Vantage Point tree (VP-tree)
- Locality Sensitive hashing (LSH)
- Prefix tree with clusterng (Prefix-tree)

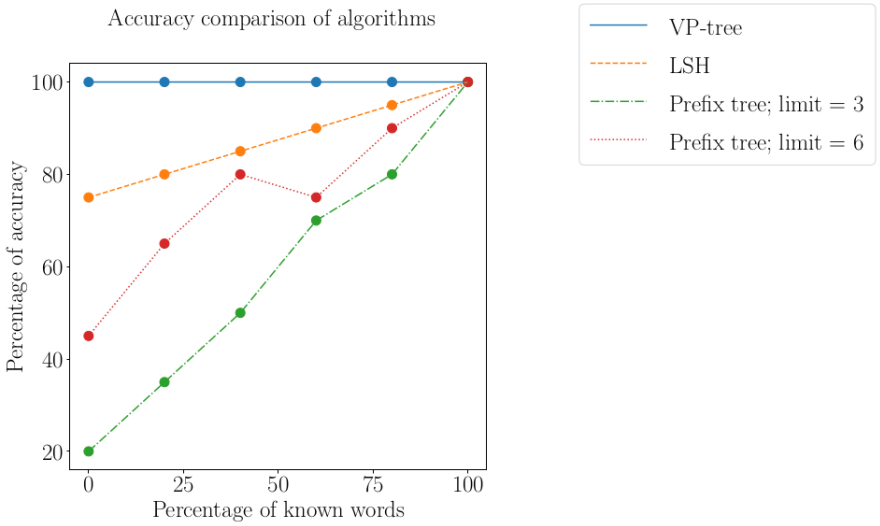

Figure 5. Accuracy comparison

VP-tree and LSH are more conventional and widely known methods, while Prefix-tree is a more recent approach. We adapted each algorithm to a uniform framework to perform a range of tests. From the results of our tests we can conclude the following:

- LSH takes the less time to build up.
- VP-tree is the most accurate of the three methods.
- Prefix-tree and LSH clearly outperform VP-tree in regards of search time.
- The speed comparison between Prefix-tree and LSH is not one-sided, but Prefix-tree scales better, while LSH is less affected by whether the searched word is inside the structure or not.
- Each algorithm scales reasonably depending on the k value at k-NNS operation.
- The difference between insertion speed of the three algorithms is rather small, Prefix-tree being slightly slower on the evaluated interval.

6. Acknowledgements

The described article was carried out as part of the EFOP-3.6.1-16-2016-00011 “Younger and Renewing University – Innovative Knowledge City – institutional development of the University of Miskolc aiming at intelligent specialisation” project implemented in the framework of the Szechenyi 2020

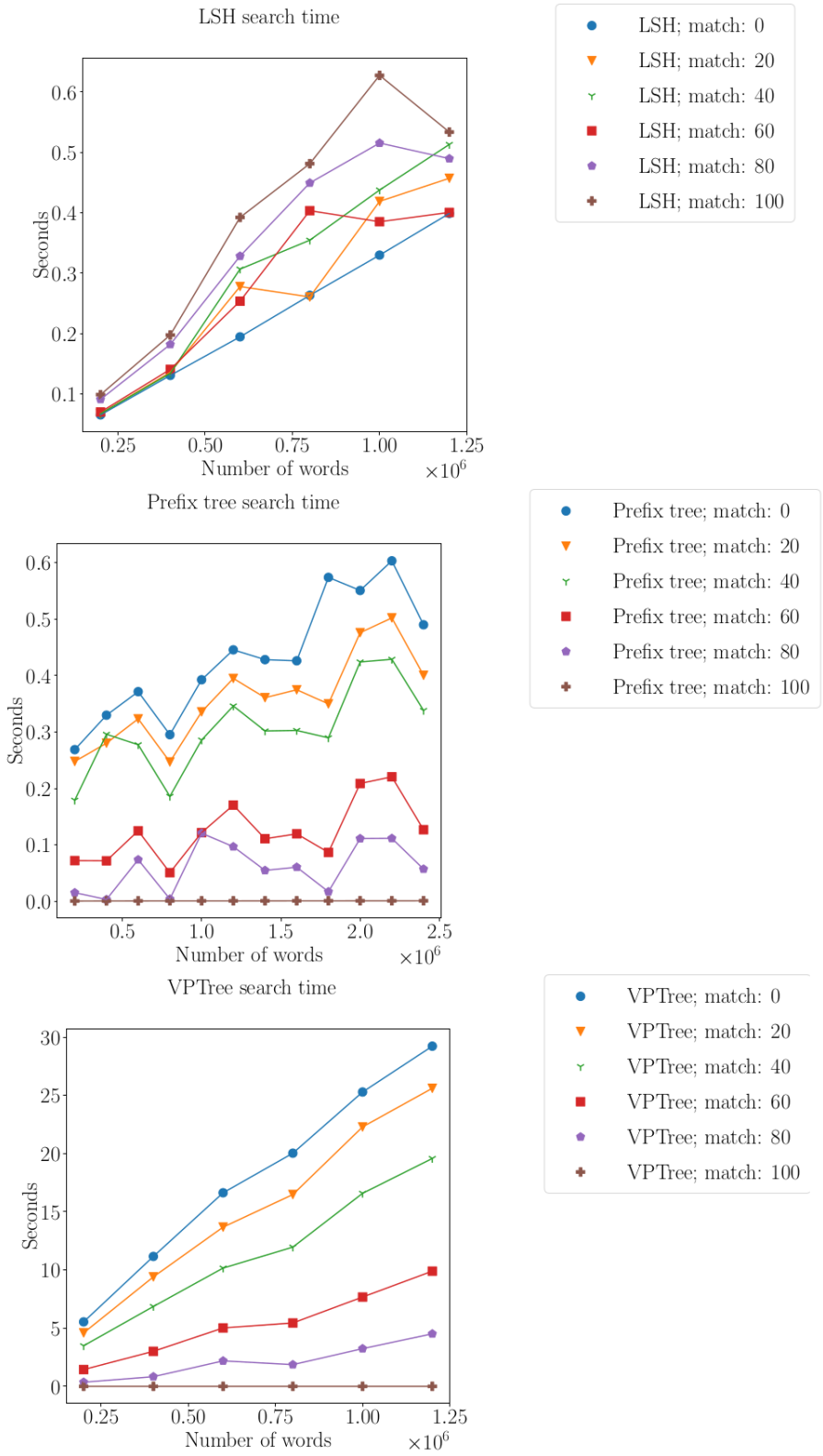

Figure 6. KNN search time comparison

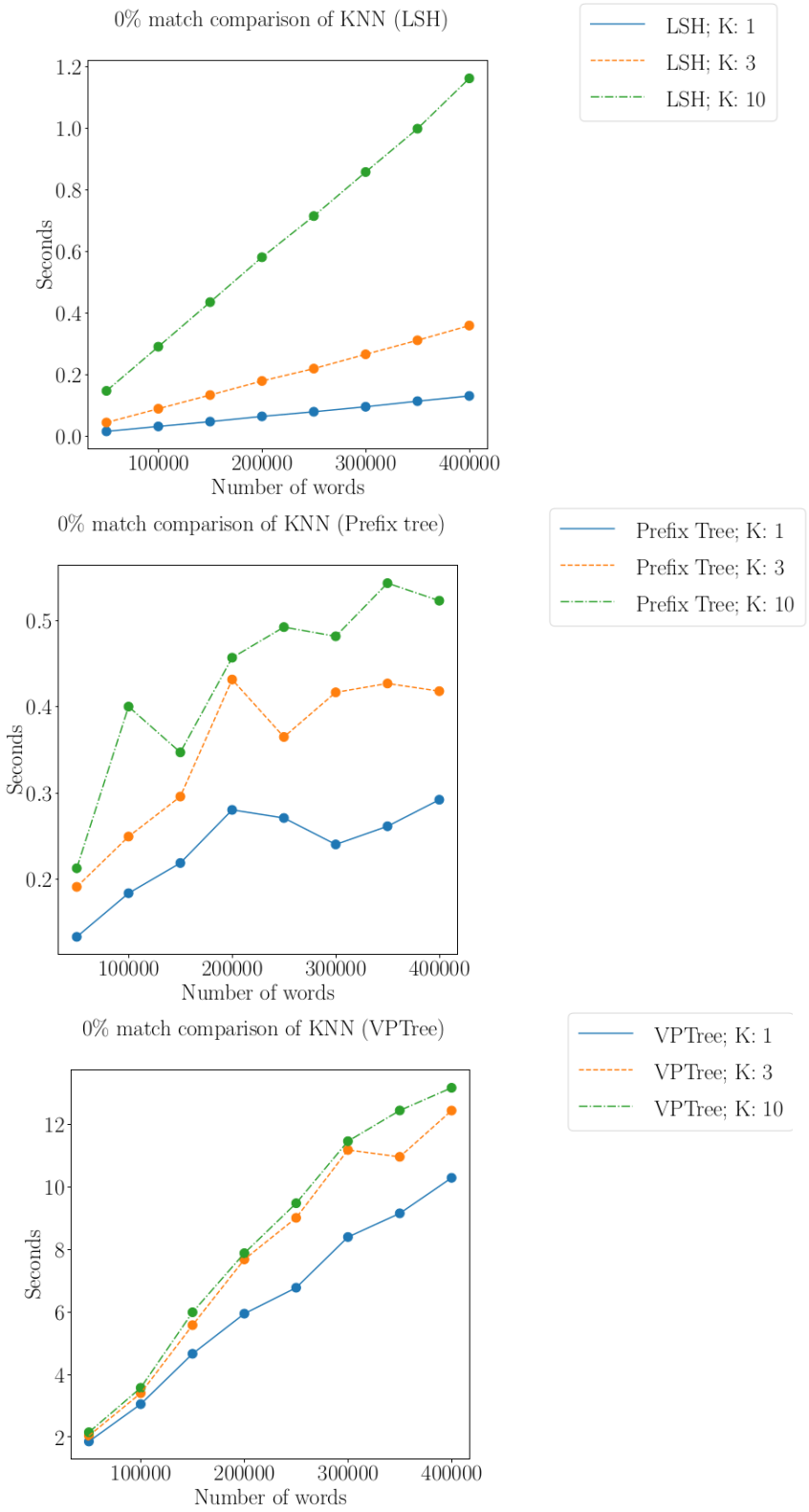

Figure 7. KNN search time comparison

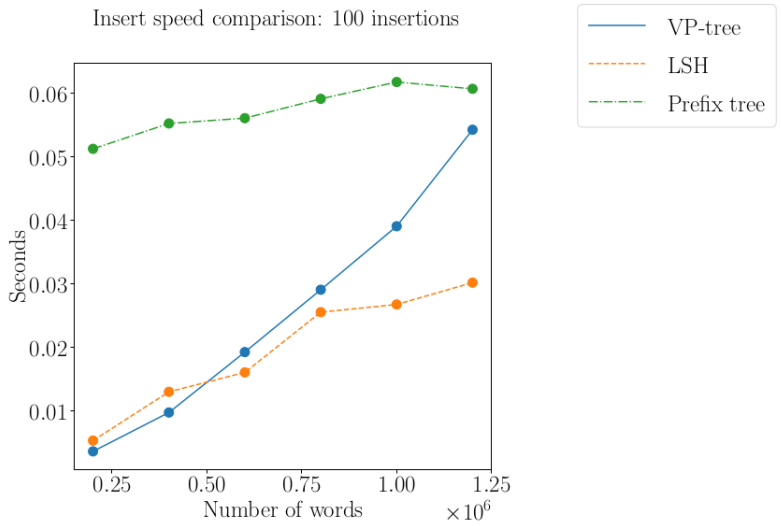

Figure 8. Insert comparison

program. The realization of this project is supported by the European Union, co-financed by the European Social Fund.

REFERENCES

[1] YULIANTO, M. M., ARIFUDIN, R., and ALAMSYAH, A.: Autocomplete and spell checking levenshtein distance algorithm to getting text suggest error data searching in library. *Scientific Journal of Informatics*, **5**(1), (2018), 75.

[2] AGRAWAL, M. and SHARMA, D. K.: A state of art on source code plagiarism detection. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, IEEE, 2016, pp. 236–241.

[3] BABA, K.: Fast plagiarism detection using approximate string matching and vector representation of words. In *Behavior Engineering and Applications*, pp. 67–79, Springer, 2018.

[4] SRIVASTAVA, S., MUKHERJEE, P., and LALL, B.: implag: Detecting image plagiarism using hierarchical near duplicate retrieval. In *2015 Annual IEEE India Conference (INDICON)*, IEEE, 2015, pp. 1–6.

[5] POTHARAJU, R., NEWELL, A., NITA-ROTARU, C., and ZHANG, X.: Plagiarizing smartphone applications: attack strategies and defense techniques. In *International symposium on engineering secure software and systems*, Springer, 2012, pp. 106–120.

[6] HUSSAIN, S. F. and SURYANI, A.: On retrieving intelligently plagiarized documents using semantic similarity. *Engineering Applications of Artificial Intelligence*, **45**, (2015), 246–258.

[7] Goyal, A., Daumé III, H., and Guerra, R.: Fast large-scale approximate graph construction for nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012, pp. 1069–1080.

[8] Liu, T., Rosenberg, C., and Rowley, H. A.: Clustering billions of images with large scale nearest neighbor search. In *2007 IEEE workshop on applications of computer vision (WACV'07)*, IEEE, 2007, pp. 28–28.

[9] Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E., and Protopapas, Z.: *Fast nearest neighbor search in medical image databases*. Tech. rep., 1998.

[10] Fu, A. W.-c., Chan, P. M.-s., Cheung, Y.-L., and Moon, Y. S.: Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal*, **9**(2), (2000), 154–173, URL `https://doi.org/10.1007/PL00010672`.

[11] Carter, J. L. and Wegman, M. N.: Universal classes of hash functions. *Journal of computer and system sciences*, **18**(2), (1979), 143–154.

[12] Gionis, A., Indyk, P., Motwani, R., et al.: Similarity search in high dimensions via hashing. In *Vldb*, vol. 99, 1999, pp. 518–529.

[13] Rajaraman, A. and Ullman, J. D.: *Mining of massive datasets*. Cambridge University Press, 2011.

[14] Kovacs, L. and Szabó, G.: Automated learning of the morphological characteristics of the Hungarian language for inflection and morphological analysis.

[15] Sjögren, R.: VP-Tree. URL `https://github.com/RickardSjogren/vptree`. Undefined: undefined Copyright 2017 Rickard Sjögren Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.