

DYNAMIC DICTIONARY WITH CONDITIONAL PERFECTLY BALANCED BINARY SEARCH

Vladislav Vasilev, Georgi Iliev

Faculty of Telecommunications, Technical University of Sofia, 1000 Sofia
Bulgaria

* Corresponding Author, e-mail: gli@tu-sofia.bg

Abstract: In this paper we derived a novel dynamic dictionary set of algorithms that support perfectly balanced binary searches asymptotically. The dynamic dictionary is part of our FSP_vgv open-source C# package that aims to implement a portable version of Octave/Matlab in order to enable faster iterative design. Our package does not attempt to outdate other software, but fills specialized needs listed in this work. By processing the version control commit history of the FSP_vgv package we validate empirically that with unknown research horizon the time spent developing grows exponentially in the volume of production code. We identify an exponential growth parameter related to how intuitive a programming language is which also controls the research time hence proving the need for highly intuitive and portable languages.

Key words: dynamic dictionary, perfectly balanced search, iterative design, interpreter languages, open source.

1. INTRODUCTION

As a method to organize and manage data, dictionaries have been around for thousands of years which makes it one of the oldest known fields of research that is still active today [1]. The classical way to implement dictionaries are binary search trees [2] but the important drawback of most binary trees is that they have a variable height. For example, Red Black Tress has a height that is at most $2 * \log(n)$ meaning that some searches might be twice as fast with a cost of $\log(n)$. As a result, if we want to look up dictionary elements and then send those over a time sensitive channel the Red-Black tree will incur jitter even before we consider any communication sources of jitter. This particular issue gave rise to the so called Perfectly Balanced (PB) searches [3] that always take exactly $\log(n)$ steps. As a result, PB trees are highly valued for latency-sensitive applications, real-time systems and high-performance computing [4].

Historically PB searches came at higher maintenance cost. In their seminal paper [5] Stout-Warren developed a PB search but it also required global rebuilds which is not ideal for highly dynamic scenarios. As a result, many computer scientists resolve to soften the PB constraints and bridge other aspects of the searches. For example, binary

tree often ignores cache and memory hierarchy efficiency. As a result, we've seen the rise of perfectly balanced multi-way trees, such as B-Trees [6] and variants for example, B+ Trees [7] which achieve balance but also better utilization of cache and memory.

Alternatively, one way to enable better dictionary searches is to consider how the customer will use it which in turn can help simplify the algorithms. For example, in [8] it was observed that programmers use the AI in two modes: acceleration and exploration. The developers either know what they need and need to get there faster or they are unsure and use the AI to explore options. Knowing about this user pattern the AI can be fine-tuned to yield improvements in terms of speed and relevance.

2. RELATED WORKS

The algorithms we develop in this paper solve the PB dictionary problem by using prefixes, tails and binary searches. One of the earliest cited uses of prefixes in PB searches are Crit-bit Trees from this seminal paper [9]. However, the use of prefixes in dictionaries has also been heavily developed for the purposes of Chinese word segmentation. Because in Chinese, words are not separated by spaces this defines the need for segmentation for natural language processing. As a result, the first letters of a sentence must be treated as a prefix of the actual first word. The prefix is then looked up in a dictionary of possible words in order to detect the first words. After the first word detection the sentence is updated and the process repeated until the segmentation is done. Interestingly in some cases [10] authors use hash tables instead of binary trees. Another interesting use of prefixes in dictionaries is for keystroke minimization, where the purpose is to predict the whole word that will be typed by looking up the prefix of the word based on the first few keystrokes. For example, in [11] prefix look up in B+ tree is used to minimizing the efforts of Braille users while interacting through Braille keyboard. In this context our contribution is a special prefix transformation and a theorem that makes sure that words placed in the dictionary go in different tails which produces the PB property.

As we describe in the next section our particular PB dictionary aims at enabling an interpreter function to manage a large set of variables in order to implement a higher-level programming language. This in turn enables a significant Iterative Design (ID) speed up which has had an indisputable historical importance [12]. The straightforward approach to speed up of ID is to improve the resource management [13], where the aim is to create software with minimal negative consequences and maximizing positive contributions. Alternatively allowing more innovators to approach the problem has been a huge ID speed up and its success has given rise to open research platforms that foster collaboration, interoperability, and scientific reproducibility [14].

One particularly difficult task that deals with ID is its measurement. There is no generally accepted way to quantify and compare one ID methodology to another and as a result some researchers [15] focus of defining a formula that measures ID. Unfortunately obtaining ID data to actively estimate such measure is practically impossible for two reasons. Firstly, the software market is highly competitive and any shared ID data would reveal important inner workings that can be used against the source. Secondly once the ID data is shared it no longer adheres to scientific

reproducibility because the ID was done with unknown final result while the reproducing experiment will have at least some prior knowledge of the end result and milestones. As a result, the open-source repository defined below can also help enable ID measurement research.

3. DESCRIPTION OF THE PROBLEM

Because the modern age has seen rapid developments in big data and machine learning then we've seen a degree of standardisation of even more complex solutions like neural networks and dictionary algorithms. As a result, the active areas of research and development have entered use cases that are extremely specialised. This implies a paradigm shift. Because the problems at hand are niche and highly specialised then the problem formulations and deliverables will be vaguer and the potential results are less likely to be positive. This directly implies that the monetary return is less likely with higher individual milestone risks. In addition to that higher specialization suggests it will be more difficult to find researched able to work on the particular problem due to the amount of domain specific knowledge requirements.

If such a situation arises the researcher will need access and knowledge to the inner workings of the carrier services, which can only happen if it is open source, relatively small in size to be humanly possible to study within short time frames and extremely comprehensible to avoid mental burn out. The immediate solution to this paradigm shift is that there will be more need for the ID paradigm to be applied by individual innovators meaning single research will have to be responsible for the entire toolchain of their own solution.

The proposed set of algorithms that address the problems described here is referred to as the FSP_vgv package that is implemented in C# and the latest version can be found in this [16] repository. In this research context the FSP_vgv package aims at allowing individual researcher to apply ID by means of ease of access, lower start-up costs, intuitive syntax and comments.

Additionally, software libraries that run PB searches and are free to use on any operating system are not trivial to find. For example, red-black trees are not implemented in the system namespace of C#. The FSP_vgv only import the system namespace of C# and so the package compiles on Windows, Linux and on Android though the use of the Termux app.

Furthermore, online open-source libraries often do not contain enough comments, documentation, examples and test procedures that allows a user to maintain them should a mission critical issue arise. The FSP_vgv is commented and developed in a style similar to Andrew Ng's Machine Learning course on the coursera.org platform that did its program exercise in Octave. The FSP_vgv has at least one comment for almost any important code line, has built in test functions and examples, contains its own documentation and has a function that computes and displays its function definitions. Next, to simplify the compile command in any command line interface the FSP_vgv is a single text file maintained with the code version features of Bitbucket. As a result, even a minimal freely available SDK should be able to compile the package on any OS and on any device simply because there are no external libraries, namespaces and compile

settings. This also means that the compiled executable of the package is extremely portable only around 2-3MB compared to Octave and Matlab which are GB in size.

Surprisingly we can be more exact in the study of ID for software development. Realistically ID coding with unknown horizon can be stated in the context of dynamic programming or constraint satisfaction problem which is a well-known NP-complete problem. However to evaluate the efficiency of the programming language itself we'd need to define the system in such a way that we obtain an empiric measure of individual code expression given the entire volume of code and its growing utility during development. To this end, we propose the following model.

Assume that there is a volume of code that solves a niche and highly specialised problem for which we have not standard solution, or if it is too elaborate to retrofit an existing solution to the problem. Let a volume of code x take time T to be coded in a language L giving $T = L(x)$. Consider then adding dx lines of code to x resulting in $T^{new} = L(x + dx)$. Consider the meaning of the following derivative $[T^{new} - T]/dx = [L(x + dx) - L(x)]/dx$. From a practical perspective when we use code to search for a solution, we make singular attempts to solve the problem at hand (also referred to as iterations) and at the end of the attempt we examine what we've learned, redesign, recode and prepare for another attempt. In between attempts it is highly likely that we need to completely delete or edit a large portion $a \in [0, 1]$ of our code since we've learned what would not solve the problem and so the derivative equals the quantity $a \cdot L(x)$ in Eq. 1:

$$\frac{[T^{new} - T]}{dx} = \frac{[L(x + dx) - L(x)]}{dx} = a * L(x), \quad (1)$$

where the parameter a is a constant. The solution of Eq. 1 is unique, and it is the exponential function $L(x) = h \cdot \exp(a \cdot x)$. Additionally, $L(1) = 1 = h \cdot \exp(a \cdot 1)$ gives Eq.2:

$$T = L(x) = \frac{\exp(a \cdot x)}{\exp(a)} = \exp(a \cdot [x - 1]) \quad (2)$$

Therefore, the entire programming language L is described with a single parameter a , and it is the single most important parameter to consider at problem inception due to the exponential nature of the relation. Clearly that parameter is strongly dependent on how intuitive the programming language is. Therefore Eq. 2 also explains why we should not use C# directly but utilize the FSP_vgv interpreter as ID speed up.

4. PROPOSED SOLUTION

Importantly the FSP_vgv package [16] contains a lot of important information that could not fit in this paper like, inspiration of the theoretical solution, changes made between each version, implementation, testing, nested dictionaries, structure of dictionaries and a machine learning search algorithm for upper and lower limits based on the dictionary algorithm. In essence we are continuing the work done in [17] where binary search was performed on a sorted vector by iteratively relaxing the search interval. The relaxation is performed by comparing the search value with the middle element and then reducing the search interval to the corresponding sub-interval. Specifically, the binary searches in [17] were used for simplicial decomposition and the algorithm appeared first in [18].

To model these binary searches, in the context of object-oriented programming and more specifically C# we'll express the vector as a dynamic list and then we'll make additional construction on top of the dynamic list to achieve a dynamic dictionary. We start with notation. We only ever consider a single class and let us call it Folded Sheet of Paper (FSP_vgv). We denote each FSP_vgv instance with a capital letter and each attribute of the object with a dot. For example, $X.next$ refers to an object with name X and its attribute $next$. Furthermore, each object X has a name field $X.name$ which is a unique string in the dictionary that we use to sort, place and look up the object. Therefore, depending on the name $B.name$ might come before $A.name$ in the dictionary. Additionally, the order and structure of the dictionary is indicated with arrows, where for example $X.next \rightarrow Y$ means that next is a pointer of object X currently pointing to Y . When we express $X \rightarrow Y$ then we've denoted that there is some pointer of X that points to Y without explicitly stating which one. To construct the dynamic dictionary we define each object to have the following pointers: $X.next, X.left, X.right, X.tail$ and an example of how these are organized to produce the dictionary is given in Fig. 1.

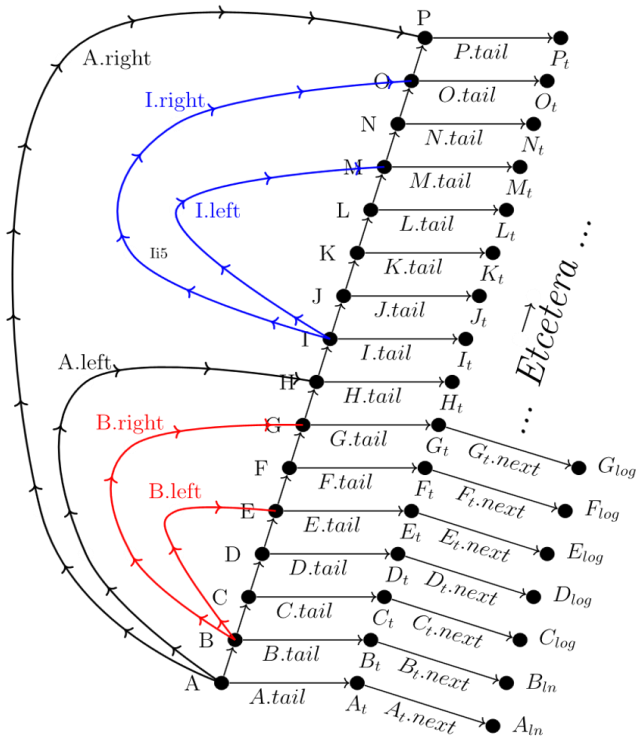


Figure 1. An example of a general FSP_vgv dynamic dictionary structure

It is worth noting at this point that the example in Fig. 1 is a standard dynamic list by means of nodes $A, B, C \dots P$ and the remaining pointers and nodes act as an additional construction to enable the list to perform dynamic edits in a logarithmic number of steps

on average. In fact the binary search on ordered elements $A, B, C \dots P$ was already defined in [17] on vectors and in this paper we further develop it for dynamic lists.

Next, observe that all nodes in Fig. 1 have an out degree of at most 4 which allows the construction to be represented with only 4 pointers. In Fig. 1. all the nodes $A, B, C \dots P$ including the ones that are pointed as tails $A_t, B_t \dots P_t$ could either contain data or are empty. We purposefully insert nodes that don't hold data but have specialized names in order to enable dynamic edits and we call these nodes phantoms. Additionally, we call main nodes all the nodes that are not in any tail which are $A, B, C \dots P$ in Fig 1. Similarly, nodes that are in a tail like $A_t, B_t \dots P_t$ are referred to as tail nodes. Let us denote the data size with n . Then the main nodes are subject to sorting and can be ordered in any desired length $O(n)$, while the tail nodes are not sorted and each tail could only contain $O[\ln(n)]$ nodes. As a result, a binary search in the sorted main nodes takes $O[\ln(n)]$ and a secondary search within a single tail takes $O[\ln(n)]$.

Now we describe how we sort the main nodes to enable binary searched on them. In both cases this is achieved by the use of the $X.next$, $X.left$ and $X.right$ pointers. First of all, the input data would be structured as a dynamic list only using $X.next$, for example $A.next \rightarrow B.next \rightarrow C.next \rightarrow \dots \rightarrow P$. Next, we apply a type of folding on the dynamic list which also derives the name of the set of algorithms. Starting at A we find the middle of the list which in this example is H and we set $A.left \rightarrow H$. Similarly, we also find the end P and set $A.right \rightarrow P$. Finding the middle and end of the list takes $O(n)$, and in essence folds the list in half. Within the two sub-lists B to G and I to O we can repeat the same folding procedure as showed in Fig. 1. As a result $B.left \rightarrow E, B.right \rightarrow G$ and $I.left \rightarrow M, I.right \rightarrow O$. Because the sub-folding halves the length of the list there are at most $O[\ln(n)]$ foldings where each fold takes $O(n)$ to a combined total of $O[n \cdot \ln(n)]$. Importantly the folded list of main nodes allows sorting with the use of merge sort. For example, if we assume list B to G is sorted and I to O is sorted then we can combine them in a list B to O at a cost of $O(n)$ steps. This is true because in any two sorted lists the smallest element all together is the first element of one or the other list. For example, the smallest element in B to G and B to O would be either B or O . Therefore, sorting with the merge sort algorithm in the folded structure also takes $O[n \cdot \ln(n)]$.

We can look up individual nodes within the sorted main nodes using the left and right pointers with a binary search. For example, assume we are looking $s = J.name$. We start at A and compare s to $A.left.name = H.name$. Because s appears after $H.name$ then we can exclude A to H and recur the check within I to P . Because after each check we halve the search interval there are at most $O[\ln(n)]$ iterations. Next, we explain how we allow dynamic edits of the entire structure through the use of tails, phantom nodes and an algorithm we call prefix decomposition.

4.1. Prefix decomposition and Phantom nodes

Assume we've already sorted the data in the main nodes so we can look up any main node within $O[\ln(n)]$. Also observe that once we look up a main node, we can do a linear scan of its tail that costs $O[\ln(n)]$ because we never allow a tail to be longer than $O[\ln(n)]$.

For the remainder of this section we use Matlab notation to express the sting operation, specifically square brackets indicate string concatenation in the order they appear for example $x = ['ab', 'c'] = 'abc'$ and with column we indicate the indexes we select for example $x(2:3) = 'bc'$.

For any lexicographically sorted dictionary we make the following observation. If $B.name$ is after $A.name$ and before $C.name$ then for $k \geq 0$ the substrings of the first k letters of $A.name(1:k)$, $B.name(1:k)$ and $C.name(1:k)$ are the same string, while the substrings with the letters after the k -th are also sorted lexicographically, meaning $A.name(k+1:end)$ is before $B.name(k+1:end)$ which is before $C.name(k+1:end)$. Let us call the substring of $B.name(1:k)$ the main prefix of B . For example if $A.name = 'precompute'$, $B.name = 'predetermine'$ and $C.name = 'prefix'$ then $k = 3$ the main prefix of B is 'pre' and $A.name(k+1:end) = 'compute'$, $B.name(k+1:end) = 'determine'$ and $C.name(k+1:end) = 'fix'$ are also sorted lexicographically.

As a result we can define the prefix decomposition as the procedure where we decompose the input word $B.name$ by looking it up in the dictionary to find $A.name$ comes before $B.name$ and after $C.name$, then skipping the main prefix and repeating until there are not letters left. As an example of prefix decomposition let $A.name = 'aa'$, $B.name = 'ba'$ and $C.name = 'bd'$ then the search word $X.name = 'aabd'$ will first find A then C and the resulting prefix decomposition is $['aa', 'bd']$. It is also important to note that when $k = 0$ then we just skip the first letter, for example the prefix decomposition of $X.name = 'aaxb'$ would visit the nodes A, B, C and then break the string into $['aa', 'x', 'b']$ since there are no words that can give a prefix of 'x'.

We can make sure that the prefix decomposition will always have $k > 0$ with prefixes matching by at least one character by introducing phantom nodes which hold no data but fill in prefix gaps in the dictionary. Let us define $d = \text{Floor}[\log_a(n)]$ as the dimension of the dictionary, where n is the data size, a are the number of characters in the alphabet and the function $\text{Floor}[*]$ returns the largest integer not greater than the input. Therefore, for two consecutive words $A.name \rightarrow B.name$ in the sorted dictionary we can consider what prefixes of length d are missing. For example, for $d = 1$, $A.name = 'a'$ and $B.name = 'c'$ then only prefix missing would be 'b'. On the other hand if $d = 2$, $A.name = 'ab'$ and $B.name = 'c'$ the prefixes that are missing are 'ac', 'ad' ..., 'az', 'b', ... 'ba', 'bd' ... 'bz' if the alphabet only contains lower case letters. For the dictionary to have no prefix gaps we might need to add phantoms between all consecutive words. Importantly by only considering adding phantoms with prefixes up to the dictionary dimension d we make sure the dictionary remains of size $O(n)$ because all possible prefixes of length d are at most $a^{\log_a(n)} = n$.

4.2. Permuted prefixes

Both the prefix decomposition and the phantom nodes were designed to solve the tail overfilling problem however to prove that we can avoid this overfilling entirely we'll need to re-examine the prefix decomposition. Observe that for an input $X.name$ the prefix decomposition will partition its name into main prefixes $X.name = [mp_1, mp_2 \dots mp_h]$ corresponding to the set of dictionary element $Q = [Y_1, Y_2 \dots, Y_h]$ that

are visited by the binary searches. Because of the one-to-one mapping $Q_i \leftrightarrow mp_i$ then if we permute the name based on the prefixes, we'll create a new name, but we'll visit exactly the same set of elements Q . For example, $W.name = [mp_2, mp_1 \dots mp_h]$ will visit the same $Q = [Y_1, Y_2 \dots, Y_h]$ but in a permuted order. As a result, if by some chance the user inserts such names that are various main prefix permutations that give the same exact set $Q = [Y_1, Y_2 \dots, Y_h]$ then these tails will get filled past the logarithmic length while the remaining tails might be completely empty. This will void the dictionary property and to deal with this situation we define permuted names.

Let's assume that we've inserted phantom nodes so that for $X.name = \text{Concat}[mp_1, mp_2 \dots mp_h]$ each mp_i is a string of length at least $k \geq 2$. Consider then the string $perm_1 = [mp_1(1), mp_2(1) \dots mp_h(1)]$ where we concatenate the first character of each main prefix mp_i in their respective order. Therefore in general let $perm_j = [mp_1(j), mp_2(j) \dots mp_h(j)]$ be a concatenation of the j th character of each main prefix mp_i in the order in which the prefixes appear in the prefix decomposition. Then let us define the permuted prefix name $X.name_perm = [X.name, perm_1, perm_2, \dots perm_k]$. As an example if the prefix decomposition is $X.name = ['ab', 'cd']$ then the permuted prefixed would be $'acbd'$ and the permuted prefix name would be $X.name_perm = ['ab', 'cd', 'acbd']$. Let us run the prefix decomposition on $X.name_perm$. The first prefixes that appear are guaranteed to be the prefix decomposition of $X.name$ meaning $X.name = [mp_1, mp_2 \dots mp_h \dots]$. Following mp_h we'll see a prefix decomposition of the remaining string $[perm_1, perm_2, \dots perm_k]$ into $[p_1, p_2, \dots p_g]$ resulting in $X.name_perm = [mp_1, mp_2 \dots mp_h, p_1, p_2, \dots p_g]$. For this prefix decomposition with permuted name, we can prove the following theorem:

Theorem 1. In a sorted dictionary FSP_vgv that has phantoms which fill all the prefix gaps performing prefix decomposition with appended permuted names will indicate such insert tails that any tail will not exceed a length of $O(\log[n])$ even if we insert $O(n)$ new words.

Proof of Theorem 1. Consider the names $name_perm^a$ and $name_perm^b$ which are different to one another in at least one character position. First assume that there is at least one main prefix v that is unique meaning $\exists v \Rightarrow mp_v^a \neq mp_v^b$. This implies that the prefix decomposition will visit at least two different tails $mp_v^a \neq mp_v^b$ and so a and b will be placed in different tails.

Alternatively assume there are no unique main prefixes and instead $mp_i^a = mp_{j[i]}^b$ for some index mapping $j[i]$. Since the original names do not help in this case consider only the remaining permuted prefixes of the name, meaning $name_perm^a(d:end) = [perm_1^a, perm_2^a, \dots perm_h^a]$. Next substitute the definition of each permutation $perm_k = [mp_1(k), mp_2(k) \dots mp_h(k)]$ which gives Eq. 3:

$$name_perm^a(d:end) = [mp_1^a(1), \dots mp_h^a(1)], \dots [mp_1^a(k), \dots mp_h^a(k)], \dots \quad (3)$$

In Eq. 3 substitute $mp_i^a = mp_{j[i]}^b$ to obtain Eq.4:

$$name_perm^a(d:end) = [mp_{j[1]}^b(1), \dots mp_{j[h]}^b(1)], \dots [mp_{j[1]}^b(k), \dots mp_{j[h]}^b(k)], \dots \quad (4)$$

In this set up the main prefixes of a and b are the same sets of strings but the permuted names are not. The only way for $name_perm^a(d:end)$ and

$name_perm^b(d: end)$ to be exactly the same is if the individual characters are exactly the same $mp_i^b(k) = mp_{j[i]}^b(k)$ for all k and all i . Therefore, consider the reindexing $j[i]$, specifically it might be that $i = j[i]$ for all but at least two indexes where $i = j[s]$, $s = j[i]$ otherwise the input names will be the same. As a result, $\exists i \neq j[i]$. If $mp_i^b(k) = mp_{j[i]}^b(k)$ for all k then the two main prefixes are the same $mp_i^b = mp_{j[i]}^b$ and the reindexing $i \neq j[i]$ has no effect which means the input names were the same. Therefore, there is at least one unique character u where $\exists u, \exists i \neq j[i] \Rightarrow mp_i^b(u) \neq mp_{j[i]}^b(u)$. As a result $\exists w, z \Rightarrow perm_w^a \neq perm_z^b$ and therefore $name_perm^a(d: end) \neq name_perm^b(d: end)$ which means that a and b will be placed in different tails.

Last but not least, consider the case where $name_perm^a(1: d - 1) = name_perm^b(d: end)$, meaning the original name of some word a is the same as the permuted prefix name of another word b . We can transform this case to the previous and use proof by contradiction. Assume that $name_perm^b(1: d - 1) = name_perm^b(d: end)$ meaning assume that the name of B is the same as its permuted prefix. This also means that $name_perm^a(1: d - 1) = name_perm^b(1: d - 1)$ which brings us to the previous case where we showed that when the main prefix decomposition is the same then $\exists u, \exists i \neq j[i] \Rightarrow mp_i^b(u) \neq mp_{j[i]}^b(u)$ meaning $name_perm^a(d: end) \neq name_perm^b(d: end)$. This is a contradiction which means that the name of B and its permuted name cannot be the same in this case or more explicitly $name_perm^b(1: d - 1) \neq name_perm^b(d: end)$. This in turns means that the original names were different $name_perm^a(1: d - 1) \neq name_perm^b(1: d - 1)$ which means the two words will go in different tails.

In summary, for a fixed a we have proven that any other b can be inserted in a different tail therefore we can insert at least n elements.

5. PRESENTATION OF THE EXPERIMENTAL RESULTS AND DISCUSSION

Because we have used a repository to manage the iterative design, we can empirically validate Eq. 2 for the code of the FSP_vgv package. As a result between all two consecutive versions of the package, we know how many lines of code were removed and how many were added as well as the total number of lines.

In reality the volume of edits is much larger. First of all, the data does not count lines that were moved. Secondly the data does not count lines edits within a version. For example, we might add 100 lines and remove the same 100 lines within a version to run an experiment and then commit the new changes, but the final edit count will not have these 100 lines because we completely undid the change just before the commit. Therefore, in reality the code edits are fractal in nature, meaning the number of edits vary depending on the measurement units. For example, if we saved commits twice as often the number of edits will probably be more than twice as many.

In Fig. 2 we see the exponential pattern of Eq. 2 emerges. In Fig. 2 we also see we have produced 20k lines of production ready code for 200 versions and since the growth is seemingly linear then we have been adding 100 lines per version. On the other side for the same 200 versions, we have edited almost 80k lines of non-production code

meaning we've deleted 3 times more code that we have produced. Importantly within the first 100 versions we have edited 30K lines of code to obtain 10k lines of production code meaning we've deleted 2 times more code than we have kept. Observe that $a = 0.0126$ means we are deleting around 1.3% of the code per version which translate to a single line per version. Even this low value for a is enough to make sure that between version 100 to version 200 the percentage of code we have deleted grows from 66% to 75%. By the exponential ratio of the system the last percentage will keep rising.

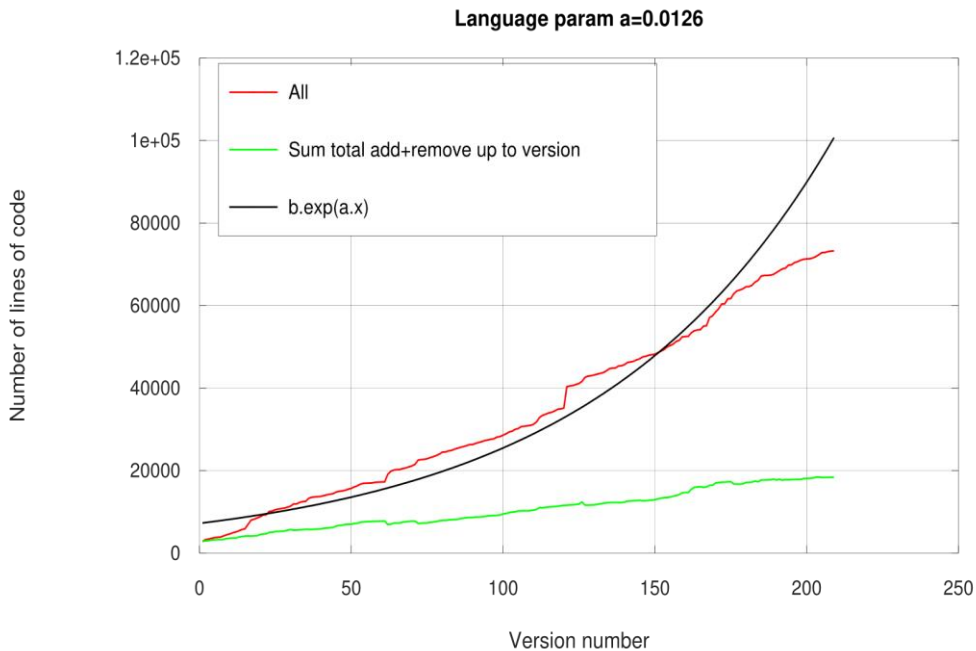


Figure 2. Empirical validation of Eq. 2

These statistics would be worse with less intuitive language, for example C++. It would be difficult to know exactly how much worse it would be because the current version control was performed with unknown horizon. If we were to recode the algorithms in another language, then we will be following the specifications of the current FSP_vgv package, and the exponential pattern will likely not occur. That is why training developers to code up well-established algorithms from scratch is vital as it trains them to make crucial ID decisions for unknown algorithms.

6. CONCLUSION

In this paper we propose a novel dynamic dictionary set of algorithms we implemented in the FSP_vgv package where the user can insert, delete and look up on average with a cost $O(\log[n])$. We have proven the asymptotic performance of the algorithms in Theorem 4.1. As we saw the algorithms rely on a structure that has two parts the main FSP_vgv and its tails. We defined the domain of utility for the FSP_vgv

package to address problems that require ID, code edit flexibility both on low and high level of coding and abstraction. We have analysed the ID approach using the repository's difference features and concluded that the number of code edits grow exponentially as the production code grows linearly. This validates the need for solutions such as the FSP_vgv empirically, where we develop with unknown horizons and limited specifications. In this process we've also shown the limitation of the software, namely for use cases that do not require custom tuning the highly optimised off-the-shelf solutions would be a better choice. Because of this future work consists of integration of the FSP_vgv package with other well established software and protocols. One particularly interesting future use would be for sensor networks or swarms of drones that have limited compute power and may require excessive fine tuning depending on deployment.

ACKNOWLEDGEMENT

The authors acknowledge the significant support of EU project: BG-RRP-2.004-005 including financial, administrative and creative uplift.

Special thanks go to Cyril Murphy without whom the development of this dynamic dictionary would not have been possible.

REFERENCES

- [1] A. Klosa-Kuckelhaus, F. Michaelis, H. Jackson. The design of internet dictionaries. *The Bloomsbury Handbook of Lexicography*, vol. 1, 2022, p. 405.
- [2] T. Cormen, et al. *Introduction to algorithms*. MIT press, 2022.
- [3] H. Lim, H.-G. Kim, C. Yim. IP address lookup for internet routers using balanced binary search with prefix vector. *IEEE Transactions on Communications*, vol. 57, no. 3, March 2029, pp. 618-621. doi: 10.1109/TCOMM.2009.03.070146.
- [4] A. Shah. A. H. Tewfik. Perfectly balanced binary sequences with optimal autocorrelation. *14th IEEE International Conference on Electronics, Circuits and Systems*, Marrakech, Morocco, 2007, pp. 681-684, doi: 10.1109/ICECS.2007.4511083.
- [5] Q. F. Stout, B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, vol. 29, no. 9, 1986, pp. 902-908.
- [6] E. D. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.
- [7] E. D. D. M. A. Bender, M. Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, vol. 35, no. 2, 2005, pp. 341-358.
- [8] S. Barke, M. B. James, N. Polikarpova. Grounded Copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, 2023, pp. 85-111.
- [9] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, vol. 15, no. 4, 1968, pp. 514-534.

- [10] Ye, Junmin, Songsong Li, Guangquan Hao, Shizi Li, Yan Yang, Cong Jin. The prefix and suffix query of Chinese word segmentation algorithm for maximum matching. *2011 International Conference on Image Analysis and Signal Processing*, pp. 74-77. IEEE, 2011. DOI: 10.1109/IASP.2011.6109001
- [11] Shrawankar, Urmila, Bharat Kapse. Prefix matching for keystroke minimization using B+ tree. *8th International Conference on Computer Science & Education*, IEEE, 2013, pp. 126-131. DOI:10.1109/ICCSE.2013.6553896
- [12] C. Larman, V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, vol. 36, no. 6, 2003, pp. 47–56.
- [13] Y. Klinchova, E. Shaban, N. Kasakliev, E. Somova. Energy efficiency of the web design based on website performance measurements. *International Journal on Information Technologies and Security*, vol. 17, no. 3, 2025, pp. 25-36. DOI: <https://doi.org/10.59035/TDLL2550>
- [14] Memeti, A., Cipi, E., Rexhepi, H., Kamberi, A. Modular architecture for open research platforms: A case study of RARS with POCOS, data mappers, and repository patterns. *International Journal on Information Technologies and Security*, vol. 17, no. 4, 2025, pp. 3-14. DOI: <https://doi.org/10.59035/CHJS9821>
- [15] Manzano, Martí, Cristina Gómez, Claudia Ayala, Silverio Martínez-Fernández, Prabhat Ram, Pilar Rodríguez, and Marc Oriol. Definition of the on-time delivery indicator in rapid software development. *2018 IEEE 1st International Workshop on Quality Requirements in Agile Projects (QuaRAP)*, pp. 1-5. IEEE, 2018. DOI:10.1109/QuaRAP.2018.00006
- [16] V. Vasilev. Folded sheet of paper. *Bitbucket*, March 2024, self-published. URL: https://bitbucket.org/doublevvinged-vladi/fsp_vgv/ (Visited on 15.02.2026).
- [17] V. Vasilev. Algorithms and Heuristics for Data Mining in Sensor Networks. LAP LAMBERT Academic Publishing, 22 December 2016.
- [18] V. Vasilev. Chromatic polynomial heuristics for connectivity prediction in wireless sensor networks. in *ICEST 2016*, Ohrid, Macedonia, 28-30 June 2016.

Information about the authors:

Vladislav Vasilev has a PhD from Technical University of Sofia, Department of Communication Networks and he is currently a research fellow at the same department. His research interests include but not limited to telecommunications, hyper-graph theory, machine learning, statistics and iterative design.

Georgi Iliev is Professor at Faculty of Telecommunications, Technical University of Sofia. His interests are in the area of communication networks, adaptive signal processing and information theory.

Manuscript received on 19 February 2026