Analysis and Implementation of Long Pattern Matching Approaches

Ferenc Galkó

Department of Automation and Applied Informatics,
Budapest University of Technology and Economics,
Műegyetem rkp. 3
Budapest, 1111, Hungary
ferenc.galko@gmail.com

Supervisor: Sándor Juhász

Department of Automation and Applied Informatics,
Budapest University of Technology and Economics,
Magyar Tudósok krt. 2 QBF207
Budapest, 1117, Hungary
juhasz.sandor@aut.bme.hu

2014

Abstract

Suffix arrays and suffix trees are well-known for their capability of efficiently solving string processing problems including exact string matching, which has many uses in a variety of fields like computational molecular biology and search engines. In this paper we present a novel way to use hash tables for exact string matching as well as our detailed comparison of the different approaches, throughout carefully selected test suites ranging from proteins to English texts. Our experimental results show that in many areas our hash table based version outperforms even the best known suffix array and suffix tree based solutions, thus indicate that this approach is not only of theoretical interest.

Contents

1 Introduction

In a rapidly developing world one can hardly argue about the growing importance of information processing and data mining. Exact and inexact string matching algorithms were one of the main focuses during the last decades, both of them have ample uses in bioinformatics, search engines, natural language processing and even in intrusion detection.

Since the exigent need for faster and faster string matching algorithms is yet to be satisfied, in this paper we present a novel way of using hash tables for exact string matching - which in some cases clearly outperforms the existing solutions - as well as compare different methods to shed light on the advantages and drawbacks of the different approaches.

Notwithstanding with the simple definition of the exact string matching problem, that is, we would like to find every occurrence of a pattern in a given text, the performance of an exact string matching algorithm is crucial, since in many cases operating on enormous amount of data is inevitable.

Allowing for the fact that many application inherently uses multiple searches with different patterns in the same text, we can preprocess the original text to facilitate multiple consecutive queries. That is to say, we create a data structure from the original text, and in the future we use both the newly constructed data structure and the original text for a possibly more efficient pattern matching.

The idea of preprocessing is already widely used, especially in the form of suffix trees and suffix arrays and their extensions, yet many scientists are still working on more efficient solutions in terms of execution time and memory usage.

Both suffix trees and suffix array are versatile data structures with many uses in string processing, and once they are created it is possible to match any pattern to the original text in linear time to the length of the pattern (and of course the number of occurrences of that particular pattern)[1]. Therefore as soon as one of these auxiliary data structures is constructed, the length of the original text is irrelevant and the execution time is solely determined by the length of the pattern and the number of occurrences.

According to this fact, one could think that we should merely focus on the construction times of these data structures, however, in many cases we

[1]Although with the suffix array alone the linear time pattern matching is not possible, it is achievable by constructing accessory data structures along with the suffix array [3].

would like to match millions if not billions of patterns to the same text, and in these cases practical results evince a thousandfold discrepancy in pattern matching times between these mathematically equally good data structures.

More than that, if the construction time of a data structure were greater than another data structure, but the first one provided a faster way for pattern matching than the latter, one could easily determine the minimum number of patterns where from the first method would be more efficient in terms of total execution time (including both construction and matching times).

Because of this, we discuss the construction and the pattern matching times independently from each other, throughout meticulously selected test suites ranging from XML files to DNA sequences, which possibly provides greater insight into the different approaches.

In this paper we suggest a new method that uses hash tables to quickly find smaller subsequences of the original pattern, thereby allowing a faster exact pattern matching than any other method known by us. In addition to introducing this approach, we also provide an implementation of a hash table which is specially designed to tackle the exact string matching problem.

The rest of the paper is organized as follows: in Section 2 we introduce the notations used in this paper as well as a formal definition for the exact string matching problem. The existing approaches, to wit, suffix arrays and trees, are also outlined in this section. Our hash table based version is described in Section 3 where we discuss the construction and pattern matching possibilities of this method in different subsections. Section 4 gives advice on optimal parameter determination, not solely based on theoretical considerations but also on our empirical results. In Section 5 we shed light on important implementation details which can drastically speed up both the construction and the pattern matching processes, while Section 6 contains an in detail comparison between currently existing implementations and the hash table based version. Finally, Conclusion and Outlook summarizes our findings and proposes directions for future improvements.

2 Notations and Previous Concepts

In this section we give a formal definition to the exact string matching problem and summarize the notations used in this paper as well as briefly introduce existing approaches.

Given a string $S = s_1 s_2 \dots s_n, (s_i \in \sum)$, where s_i denotes the $i'th$ character of S and $S_{i,j}$ denotes the substring $s_i s_{i+1} \dots s_j, i <= j$, while the suffix of S from the character i is $S_i = S_{i,n}$ and $n = |S|$ denotes the length of S. Table 1 summarizes all of the notations.

Table 1: Summary of the notations and their meanings used in this document.

Notation	**Denotes**
S	original string that contains the patterns of interest
$n = \|S\|$	length of S
s_i	character in S at the position i
$S_{i,j}$	substring $s_i s_{i+1} \dots s_j, i <= j$
$S_i = S_{i,n}$	suffix of S from position of i
S_{e_i}	string associated with the i'th edge in a suffix tree
e_i	length of the i'th edge in a suffix tree
P	pattern string
$\|P\| = m$	length of P
$\|\sum\|$	size of the alphabet
k	number of buckets in the hash table
q	special hash table parameter (see in Section 3)
$B[i]$	the i'th element of the bucket

Definition 1. *Exact String Matching: Given an alphabet* $\sum = \{a_1, a_2, \dots, a_l\}$ *and a string* $S = s_1 s_2 \dots s_n$ *in that alphabet. An exact string matching of a pattern* $P = p_1 p_2 \dots p_m, (p_i \in \sum)$ *to* S *determines all of the* $i_1, i_2, \dots, i_j$ *indexes where* $s_{i_k} = p_1, s_{i_k+1} = p_2, \dots s_{i_k+m-1} = p_m, k \in \{1, 2, \dots, j\}$.

The practical implication of this formal definition extremely simple: we have to locate every occurrence of a given pattern in a text. Of course if one of the suffixes of the pattern is also a prefix of that particular pattern, it is possible that the matches overlap.

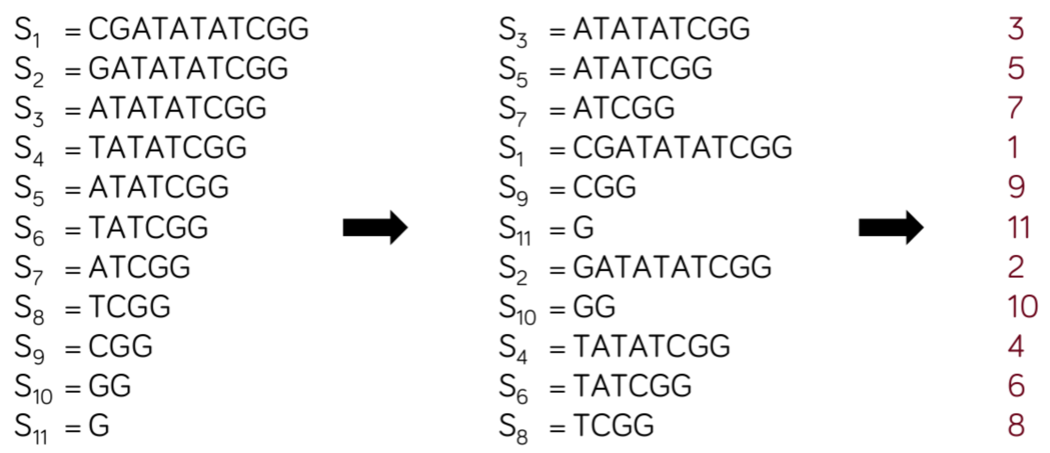

Figure 1: Example of suffix array construction for CGATATATCGG. It is sufficient to store the n (in this case 11) offset integers in the suffix array.

Thus, the result of matching $P = ATAT$ to $S = CGATATATCGG$ would be $\{3, 5\}$ while $S_3 = ATATATCGG$ and $S_5 = ATATCGG$.

2.1 Suffix Arrays

A suffix array is a simple and elegant, but nevertheless versatile data structure, which in many cases outperforms suffix trees (which is described in Section 2.2) in execution time and in memory usage. A suffix array is a sorted array that contains all of the suffixes of a string S, therefore the size of a suffix array is always $|S| = n$. Rather than storing the exact suffixes in the array, it is sufficient to store their starting indexes in the original string, which means a memory usage of $4n$ (in this paper we suppose that 4 byte pointers or indexes are sufficient to handle the size of the input strings, which is practically always the case). Figure 1 shows an example for $S = CGATATATCGG$.

Even with a simple sorting algorithm like merge sort, the sorting of the n suffixes could be done in $O(n^2 logn)$ (by comparing $O(nlogn)$ suffixes and each comparison requires $O(n)$ time), therefore the construction of a suffix array SA could also be done in that time.

In Figure 1 we illustrate the construction of a concrete suffix array for $S = CGATATATCGG$. During the first step we create a new array with the suffixes, and since the original string is already in the memory even for the first step it is sufficient to store the indexes rather than the suffixes

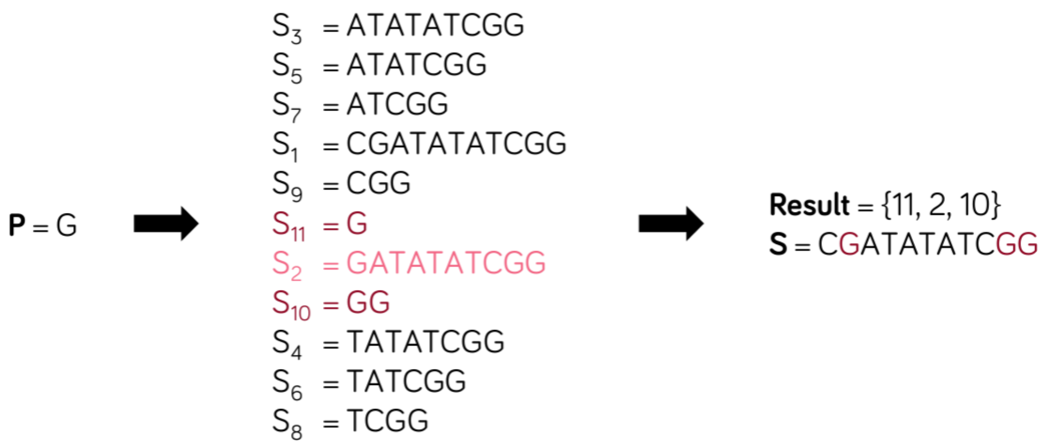

Figure 2: Example of suffix array based pattern matching for $S = CGATATATCGG$ and $P = G$.

themselves, which results in a simple sorted array of integers ranging from 1 to $|S|$. In the next step we sort the array, but instead of using the indexes, we compare the underlying suffixes during sorting.

If a suffix is a prefix of another suffix, that means that we run out of characters when comparing the first suffix to the second and still cannot decide the lexicographical order. In this case we say that the shorter one is lexicographically smaller than the larger one. In other words we put an imaginary \$ character to the end of S that is lexicographically smaller than every other character in the alphabet and then the sortation becomes unequivocal (the concept of the \$ character is widely used when discussing suffix trees and suffix arrays). In the case of Figure 1 a good example for this is S_{11} and S_2 where the previous is the prefix of the latter.

After we have a suffix array, we can perform two binary searches to determine a first i and a last j (where $i <= j$) index, so that $SA[i]$ and $SA[j]$ suffixes start with the given P pattern. Since the suffix array is sorted, each $SA[k], k \in \{i, i+1, \ldots, j\}$ suffix array entry denotes a matching suffix.

In Figure 2 we show the process of matching the simple pattern $P = G$. Firstly, we determine the first array index where the underlying suffix starts with P, in this case SA[7] (which refers to S_{11}). Then, we conduct another search to determine the last array index with the same condition, which results in the first and the last index of 7 and 9, respectively. Drawing on the fact that the suffix array is sorted we can determine these indexes with binary searches, but what is even more important is that every array index between these two denotes correct results, which means that SA[7], SA[8] and SA[9] are the solution of the problem (which refer to S_{11}, S_2 and S_{10}, respectively). In this example only SA[8] was inferred, but if there are z occurrences for a pattern, we only conduct two binary searches and infer the remaining $z - 2$ matches from those.

There are several ways to improve the execution times. With two additional arrays the desired $O(n)$ construction and the $O(m + z)$ matching can be achieved with the memory usage of $6n$ [5, 3], while by considering other practical facts the matching time can be reduced further [1], and by compressing the array a significant amount of memory can be saved [11].

2.2 Suffix Trees

A suffix tree is a data structure designed to facilitate string operations such as exact and inexact string matching (the latter one is out of the scope of our current topic) or determining the longest common substring in two strings [6].

Given a string $S, |S| = n$ there are exactly n leaf nodes in a suffix tree, and each edge denotes a substring of S. Thus, each path from the root to any leaf node determines a set of substrings, which can be concatenated into a suffix of the original string.

This means, that every internal node determines a set of suffixes that start with the prefix associated with the path to the internal node. For example in Figure 3 the leftmost internal node has two children (1 and 3) and by the time we traversed to this internal node, we know that all of these children are suffixes with the prefix of AT. Therefore, if we searched for $P = A$ or $P = AT$ we would know that there are exactly two solutions, the first and the third suffixes, without having to check any further nodes or edges.

Of course if there were another edge from the root node annotated with AT then we would leave out correct results. Luckily, this cannot happen in a suffix tree, for by definition all of the edges of a single node must start with different characters. In case of Figure 3 this does not only stand for the root node, but also for every internal node in accordance with the definition.

So how can we use the suffix tree for fast pattern matching? First, we have to get the first character of the pattern, then we have to decide which edge starts with this particular character and we shall move on the node determined by this edge. Now the first e_1 characters of the patterns are already matched correctly, where e_1 is the length of the string associated with the selected edge, because every suffix in the hierarchy starting from this nodes starts with S_{e_1} (the string associated with the edge). So we can move e_1 characters forward in the pattern and select the next edge by the character starting from that position.

At one point the concatenated length of the strings reaches the length of the pattern (formally, $e_1+e_2+\cdots+e_y >= |P|$) and we have to stop at a child node determined by this overflowing edge. Every leaf nodes in the hierarchy determined by this stopping node refers to a correct suffix, because these suffixes start with $S_{e_1}S_{e_2}\ldots S_{e_y}$ and P is the prefix of this concatenation.

In a concrete example (see in Figure 3), where $P = ATCG$, we would look for an edge starting with A from the root node, and since the edge contains $S_{e_1} = AT$, we would use the third character of the pattern during the next iteration, which is in this case a C. We arrive at the third suffix, which means that this suffix starts with the pattern, and since there are no other suffixes with this property, it is safe to say that the result set is $\{3\}$.

If we inaugurate another rule, to wit, that every internal node except for the root must have at least two children, it is easy to see that a suffix tree cannot always be constructed for a given string (a simple example is $S = AAAA$). For this, we extend the original string with a character that is lexicographically smaller than every other character in the alphabet (the \$ symbol in Figure 3) which does not effect the outcome of the queries.

Although a suffix tree can be constructed in $O(n)$ [6] time and a single pattern matching can be done in $O(m + z)$, where m is the length of the pattern to be matched and z is the number of occurrences, tree-like data structures like suffix trees are less cache friendly than array-like ones [2]. In order to build a suffix tree in linear time from a string S of length n, we have to store additional information in the nodes, thus even a careful implementation takes up to 10-20n space in memory [9].

Recent results suggest that in many, if not all applications, suffix trees can and should be replaced with suffix arrays to hasten string operations and to reduce memory usage [2].

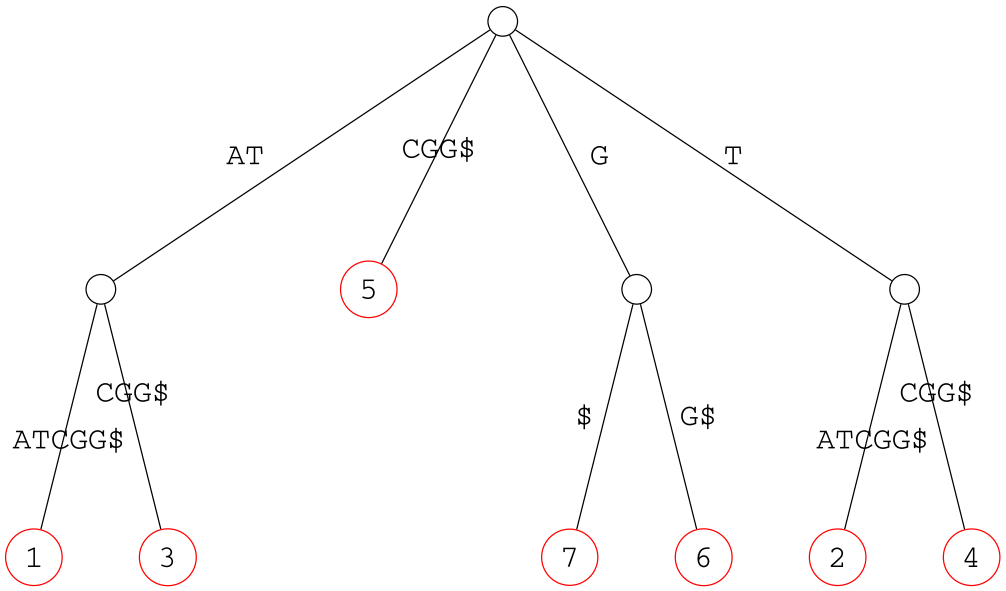

Figure 3: An example suffix tree made from $S = ATATCGG$.

3 Suffix Hash Table

Hash tables are widely used in many fields of computer science for efficiently storing key-value pairs where value retrieval of near $O(1)$ complexity takes precedence over memory usage [7, 12, 4]. In this section we introduce an algorithm based on hash tables that efficiently solves the exact string matching problem.

Hashing is already widely used for pattern matching, presented in a seminal paper by Rabin and Karp [8], and it is also possible to match multiple pattern by this existing method.

However, for this approach, the patterns must be specified and preprocessed, whereas our method does not require the set of patterns to be known, but instead it draws on the original string, in which we would like to search.

3.1 Constructing the Table

Both suffix arrays and suffix trees store suffixes of the given S string, which makes them efficient in exact string matching, for all occurrences of a given pattern are a prefix of a given suffix.

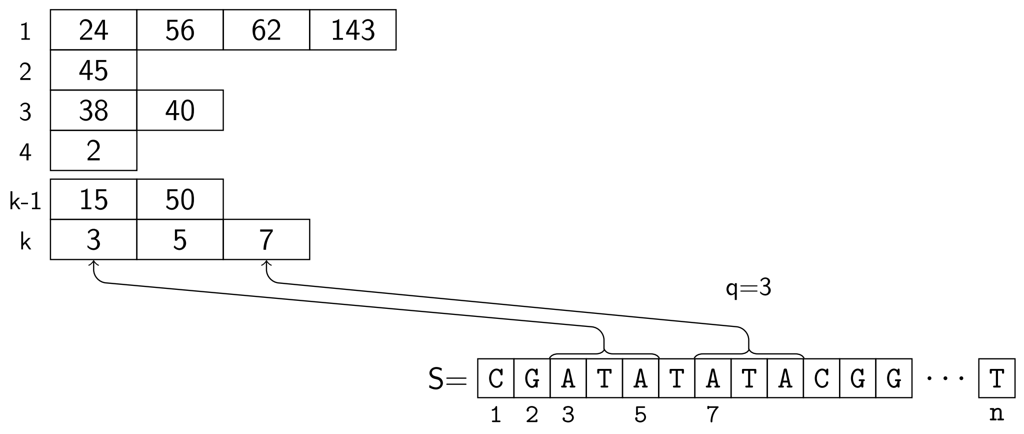

Figure 4: Storing suffixes in a hash table with k buckets while $q = 3$.

While suffix trees use trees and suffix arrays use arrays to store the suffix pointers (or offsets), hash tables could also be appropriate for the same purpose. First, we have to find a way to calculate hash values for all the possible suffixes. The idea is to generate hash values for S_i from the substring $S_{i,i-1+q}$, that is, instead of generating hash values for the complete suffix we only use the first q characters to select the appropriate bucket ($q <= n$ and in most of the cases $q << n$).

Thus, there are a total of $n - q + 1$ suffixes to be stored in a number of k buckets by using a hash function $h(x)$. Figure 4 shows and example for $q = 3$. Because the three suffix S_3, S_5 and S_7 share a common q long prefix (ATA) they certainly go into the same bucket. However, collisions are also possible since there can be one or more S_j suffixes where $h(S_{j,j-1+q}) = h(ATA), S_{j,j-1+q} \neq ATA$, thus they will be assigned to the same bucket. That is, we can never be entirely sure whether a bucket contains only suffixes with the same q-long prefixes, which we should consider during further operations.

To calculate the hash values during construction time, it is possible to use a so-called rolling hash, which is also used by the Rabin-Karp algorithm [8]. A rolling hash is a specially designed hash function, which facilitates multiple hash value generations while only two characters of the string of interest changes: the leftmost character is omitted and a new character is inserted to the rightmost position. This would mean that the hash values can be calculated independently from q in constant time.

The rolling hash, however, only hastens the construction of the table and it cannot be used efficiently during pattern matching time, which makes other hash functions possibly faster for answering queries.

As discussed in the following subsection, the hash conflicts would necessitate to check every possible result whether it is produced by hash conflict or is a real occurrence. This shall not be a problem if the average number of matches remains low, but if there were an immense amount of occurrences it would drastically slow down the process of matching. Therefore, in some cases it is advantageous to eliminate these conflicts during construction time, which although slows down the process of producing the table, provides a great speed-up during matching time. We suggest two possible ways for the hash conflict elimination, and while the first one focuses on memory usage, the second one supports faster pattern matching.

The first idea is to use a mixture of open and bucket hashing, which means that if a bucket is empty we simply put the suffix into this bucket, but if the bucket already contains an element, we compare the actual suffix with the contained suffix and on conflict (which means that the first q character of the contained suffix and the currently inserted suffix is not the same) we select another bucket for the current suffix. This way a single bucket will only contain suffixes with the same q-long prefix and we do not have to allocate any additional memory to make this possible.

However, during matching time it is possible that bucket determined by the first hash value does not contain the suffixes of interest and we have to look into further buckets, which makes this approach a memory but not pattern matching time efficient solution.

The second idea, however, optimizes - the in many cases more important - pattern matching time. During this approach, instead of finding an empty bucket for the conflicting suffixes, we split the original bucket into two or more smaller sub-buckets, each of them contains one type of suffixes. This solution is illustrated in Figure 5.

When we look up for a specific type of suffix, we have to find the corresponding sub-bucket inside the selected bucket, but as practical results attest, with well chosen parameters discussed in the next sections, we can keep the number of hash conflicts as low as one percent (which means that only one percent of the buckets would contain sub-buckets). Although the memory usage is slightly higher compared to the first solution, since all of the previously empty buckets remain unused, it provides a faster access to the suffixes.

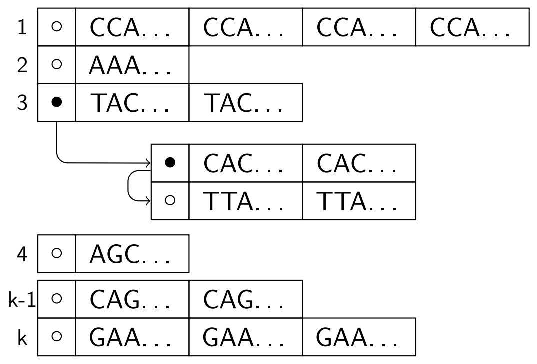

Figure 5: Storing conflicted suffixes in sub-buckets: $h(TAC) = h(CAC) = h(TTA)$.

Algorithm 1 shows a simple algorithm for building the hash table with the given q, k parameters and h hash function without considering any of the collision issues discussed above.

Algorithm 1: BuildSuffixHash(S, q, k, h)

```
$n \leftarrow len(S)$
$SH \leftarrow emptyhashtable(k)$
**for** $i = 1$ to $n - q + 1$ **do**
    store $i$ in $SH$ to bucket $h(S_{i,i-1+q})$
**end for**
**return** $SH$
```

3.2 Matching Patterns

There is a simple way to use this data structure for exact string matching. We would like to match pattern $P, |P| = m, m >= q$. First, we have to calculate the H_1 value, where $H_i = h(P_{i,i-1+q})$. If bucket H_1 contained only a small amount of indexes, we could simply check whether P is the prefix of any of the suffixes in the bucket one by one, and add the matching suffixes to the result list. In order to ensure the efficiency of the searching, we have to choose q and k (the number of buckets) of the hash table in a way that it results in an average non-empty bucket size of about one. The determination of the q and k parameters is discussed in detail in the following subsection.

Even with the best effort of choosing the above parameters there will be cases when we have to deal with buckets with many elements (for example, when there are many occurrences of the pattern string). For these cases the following method is applied to reduce the number of string comparisons: after calculating H_1 from the first q characters we retrieve all of the indexes from bucket H_1 to a *possible solutions* PS array (instead of an array we can use any kind of data structure for storing the possible solutions). Next, we should narrow down the number of candidates in array PS by checking the following q characters of the original pattern P. Instead of matching directly to the original string like the first method suggests, now we use the same hash table to retrieve the possible continuations. We take the next q characters of the pattern P and read their possible position from the hash table by checking indexes in bucket H_{1+q}. If a suffix matches the first q characters, then it is present in H_1, and if also matches the next q characters, then the continuation position should be present in H_{1+q}. For this reason we can drop all candidate positions $i \in PS$ from PS where $i + q \notin B_{H_{1+q}}$ (B_h is the bucket belonging to hash h in the hash table). If we continue this for $1 + 2q, 1 + 3q, \ldots$ we can drastically narrow down the size of PS and at the end of the process we can check the leftover indexes in PS whether they are real matches. Algorithm 2 demonstrates the described algorithm for exact string matching.

The only reason we have to check each element in PS is that there could have been hash conflicts during the creation of the hash table, which would result in erroneous results. However, it is possible overcome this issue with one of the ideas detailed in Section 3.1, which would make sure that only suffixes with a beginning of the same q characters go into the same bucket. If we constructed the hash table in accordance with this, we would not have to check all of the leftover indexes, thus a substantial amount of time could be saved during pattern matching time.

Another important note is that the buckets and possible sub-buckets are inherently sorted in an ascending order, which on we can draw for further calculations, that is, when merging the content of a bucket with PS we can use a slightly more efficient algorithm than the basic method. For this, we maintain two indexes, one for PS and one for the bucket which contains the possible continuations for the suffixes in PS.

In every iteration we increase the first, the second or both indexes by using the following rule: if $PS[i] + q = B[j]$ that means that $B[j]$ denotes a suffix that is a continuation of $PS[i]$, we keep $PS[i]$ as a possible solution and increase both i and j (since $PS[i]$ is already retained as a possible solution and it is not possible that $B[j]$ is also a continuation of $PS[i+1]$, because PS consists of different suffixes).

If $PS[i] + q > B[j]$ that means that the current element of B is not a continuation of $PS[i]$, but it is still possible that there is a correct element in the bucket, so we increase j. If $PS[i] + q < B[j]$ that means that there is no match for $PS[i]$ and we omit this element from the possible solutions array, but it is still possible that $B[j]$ is a continuation of a further element in PS so we only increase i but not j. Once we reach the end of PS with i the new possible solutions array is precisely determined, and since we increase at least one index during every iteration, the merging time is $O(|PS| + |B|)$.

Both methods require m, the length of the pattern to be greater (or equal) than q, which makes suffix hashing especially efficient for large patterns, however, there is a way to extend this data structure to be capable of dealing with smaller patterns, which we discuss in Section 3.3.

Algorithm 2: FindPattern(P, SH, h, q)

```
Require: $|P| >= q$
  $left \leftarrow len(P) - q$
  $offset \leftarrow 0$
  $PS \leftarrow SH[h(P_{1,q})]$
  while $left > 0$ and $len(PS) > 0$ do
    if left ¿= q then
      $offset \leftarrow offset + q$
    else
      $offset \leftarrow offset + left$
    end if
    for $i = 1$ to $len(PS)$ do
      if $PS[i] + offset \notin SH[h(P_{offset+1,offest+q})]$ then
        remove $i$ from $PS$
      end if
    end for
    $left \leftarrow len(P) - (offset + q)$
  end while{if needed check every element in PS to filter false results}
  return $PS$
```

3.3 Dealing with Small Patterns

In the sections above we have discussed pattern matching for patterns where $|P| >= q$. In many cases the vast majority of the patterns are over the size of q but there can be exceptions where $|P| < q$ and we still would like to search for these rare short patterns.

The problem with these patterns is that we can not search the hash table directly, because we have used q long substrings of S to store the suffixes, however with a small amendment we can refine the hash table to make these kind of searches possible.

For this we have to maintain a sorted data structure that contains pointers to non-empty buckets and sorted by the $S_{i,i-1+q}$ substrings we have used to create the hash table.

To search for the small pattern we have to do two binary searches to determine the first and the last bucket that contains suffixes starting with pattern $|P|$, these buckets and every bucket between the range of these two are possible solutions. We should keep in mind, that we have not stored the last $q-1$ suffixes of S in the hash table, therefore this method omits occurrences in those suffixes, however, we can get these results simply by searching the last q character of S for matches.

This supplementary data structure is similar to a suffix array, except that there can be a lot more suffixes than buckets therefore using a suffix array in this case can be advantageous if the maximum bucket size is not far more than 1.

4 Parameters

4.1 Parameter q and k

Since both q and k parameters play important role in this approach, we should pay special attention at their selection. A greater q could help spreading the suffixes into more buckets, while a small k ($k << n$) inevitably results in overcrowded buckets.

Our goal is to reach an average non-empty bucket size of one, and while a greater k provides more buckets to spread the elements to, increasing k beyond $2n$ does not entail substantial improvement in execution time, but wastes a lot of memory, because at least 50 percent of the buckets would certainly be empty.

To restrict the average number of elements per bucket, we have to consider that each suffix starting with the same q long substring of S goes into the same bucket. Therefore we should choose q to be big enough to provide many different values, since that can facilitate the dispersal of suffixes into different buckets. Considering this fact, the size of the alphabet $|\sum|$ also plays a key role in the determination of q.

If for example $|\sum| = 4$ and $k = n = 10^6$, then with $q = 6$ maximum a total of $4^6 = 4096$ buckets will be used, while the other 995904 will surely be totally empty. So a total of 10^6 element would be forced into a mere 4096 buckets.

However, the same q could be successfully used for a larger alphabet, for example if the alphabet size were 20, then the maximum number of used buckets would be $20^6 = 6.4 \cdot 10^7$, which combined with a good hash function and an input where the 6 long substrings of S are spread over a large domain, can be efficiently mapped to 10^6 buckets, resulting in small bucket sizes.

In Figure 6 we illustrate the average length of a non-empty bucket depending on the q and k parameters. It seems that the main determinant, as we would expect, is the q parameter, whereas k only plays a significant role in the bucket sizes when it is below $1n$, further increase in k only results in negligible bucket size reduction. Although the figure only illustrates this metric for DNA sequences, the graphs look very similar for the other types of text inputs as well (see in Section 6).

Another, equally important measurement is the number of non-empty buckets, since if there were a huge portion of buckets left empty we would allocate unnecessary space for them, thus wasting precious memory. Figure 7

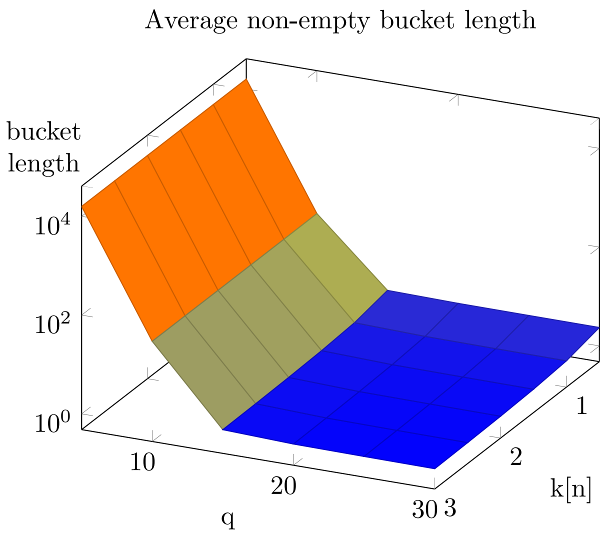

Figure 6: The average length of a non-empty bucket (calculated by using DNA sequences as inputs see in Section 6). The aim is to keep it near to 1.

shows the impact of the q and k parameters on the usage of buckets. A bigger q obviously helps to spread the suffixes into more buckets, thus increasing the number of used buckets, while a k of about $2n$ is usually a good trade-off between memory usage and efficiency.

To eliminate conflicts by using an aforementioned method (see in Section 3.1), it is important to gain insight into the number of conflicted buckets. A conflicted bucket is a bucket that contains at least 2 suffixes with different q long prefixes, so they should be stored in different buckets and the storage in the same bucket is the result of a hash conflict.

The number of conflicted buckets determines the efficiency of the methods described in 3.1, since if the number of conflicted buckets remains high, the sub-bucket solution is clearly a better approach, whereas for infrequent conflicts the mixed open and bucket hashing can provide a better solution. Figure 8 and 9 illustrate the number of conflicted buckets and suffixes, respectively.

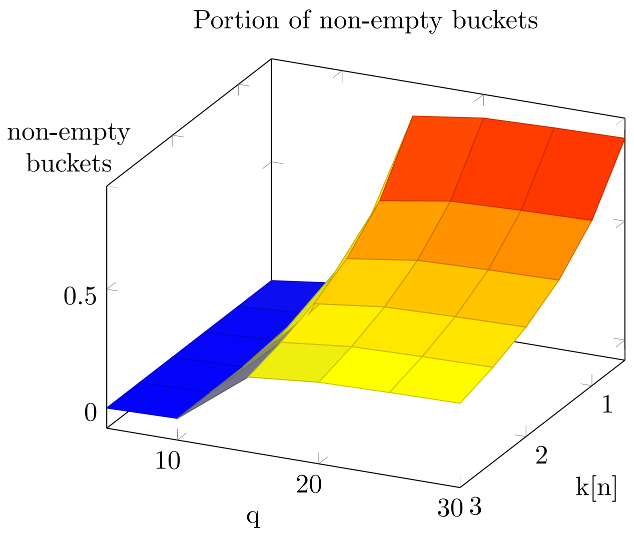

Figure 7: The portion of non-empty buckets (calculated by using DNA sequences as inputs see in Section 6).

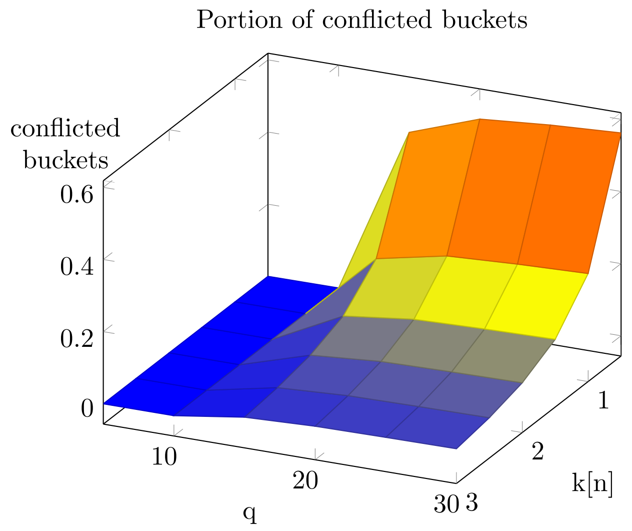

Figure 8: The portion of conflicted buckets (calculated by using DNA sequences as inputs see in Section 6).

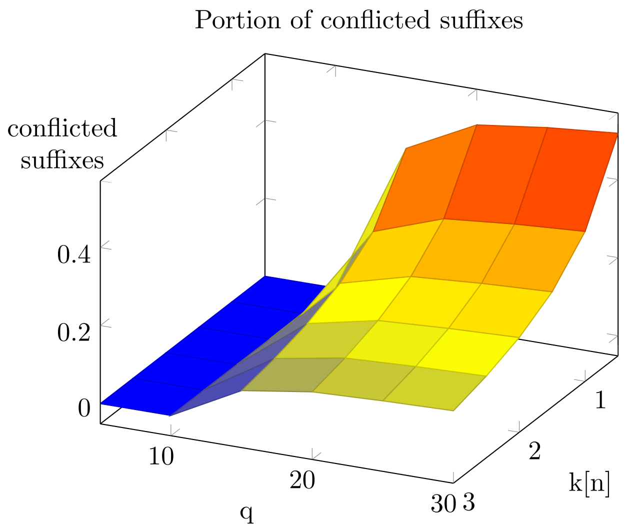

Figure 9: The portion of conflicted suffixes (calculated by using DNA sequences as inputs see in Section 6).

The bigger the q is, the faster the algorithm will find the matches. However, we should keep in mind that the algorithm cannot find matches efficiently for patterns where $|P| < q$. When searching for multiple patterns of varying sizes, we may also consider building multiple hash tables: one for a small q matching the lowest pattern size, and one for a larger q allowing to search for longer patterns efficiently.

Therefore q should be the length of the smallest pattern we would like to search for, while k should be 1-2 times n (or even more if memory efficiency is not an issue), considering that a larger k could prevent hash conflicts, but also drastically increases memory usage.

4.2 On-line Determination

Although in the current document the on-line determination of the parameters is not in the main focus, it is worth to mention that it is possible to adjust parameter k during construction time.

It would also be possible to suggest an efficient q parameter according to the concrete characteristics of the given S text, but since we cannot search for patterns smaller than q efficiently, determining a q parameter by the text and not by the future patterns is of little if no avail.

Luckily, parameter k can be dynamically adjusted to maintain an equilibrium between memory usage and matching efficiency. A simple yet easily implementable solution is to maintain a single counter of the number of empty buckets, and when the ratio of the non-empty and empty buckets reaches a threshold, we can increase (for example double) k and re-hash the suffixes.

Determining k parameter dynamically is especially useful when the length of S is not fixed, that is, we can append parts to the original string. This on-line determination can be more efficient than solely determining k by the length of S, because it also takes the specific characteristics of S into account.

5 Implementation Details

5.1 Construction

Implementing an efficient, specialized hash table for this algorithm is of great importance. Having chosen the bucket number k we should allocate memory space for k pointers, which will serve as bucket pointers.

An easily implementable yet fairly efficient solution is to use an array of k pointers as a hash table, where each pointer can point either to a real bucket or directly to a suffix in the original string. This is beneficial, because if a bucket contains one element only, then we can reach it directly (without a further memory access), but if multiple suffixes are mapped to the same bucket, then we can simply access the extendable bucket by using the pointer.

Although both are pointers, it is easy to tell suffix and bucket pointers apart: whenever we calculate a H_i hash value and check the associated bucket, we first check whether the pointer in the hash table is pointing to a substring of S or not. If the pointer is pointing to a location between the beginning and the end of S, we assume that this is the only suffix contained in this particular bucket, for we can save up a substantial amount of memory by omitting the storage of unnecessary arrays. This is especially important for properly tuned q and k parameters, where most of the non-empty buckets contain 1 element.

However, if multiple pointers are associated to a single bucket we have to store all of them in a separate array. In this case, the pointer p should be used as an array pointer and since we also have to store the size of the array somewhere, we should do it at the first position of p. This way $p[1]$ would contain the length of the array, while $p[2], p[3], \ldots, p[p[1] + 1]$ would contain suffix pointers.

The impact of the q and k parameters is illustrated in Figure 10. It is clear that both q and k play significant role in this metric of vital importance. A larger q helps to spread the suffixes into more buckets, while using more buckets reduces the number of conflicted buckets, therefore leads to more buckets with size of 1. For specific type of texts an outstanding direct storage percentage can be achieved by properly tuned parameters. For example in the case of DNA sequences, with $k = 3n$ and $q = 30$ about 70 percent of the suffixes can be stored directly in the hash table (see in Figure 10).

However, for XML files only about 40 percent can be reached, due to the q-long repetitions in the original XML input (see in Figure 11).

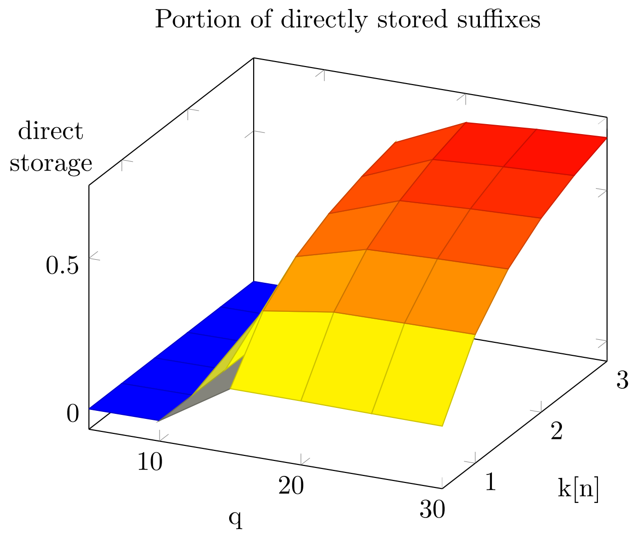

Figure 10: The portion of directly stored suffixes (calculated by using DNA sequences as inputs see in Section 6).

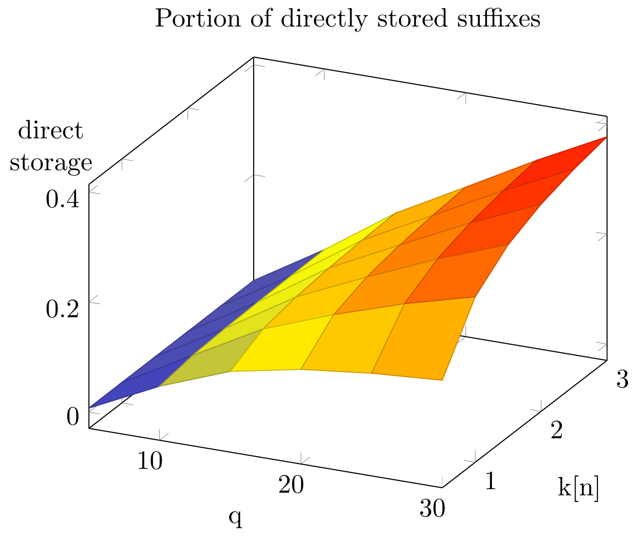

Figure 11: The portion of directly stored suffixes (calculated by using the XML file as an input see in Section 6).

The allocated k pointer requires a total of $4k$ space in memory in bytes (assuming 32-bit integers) and the worst-case scenario is when every suffix goes into the same bucket, which demands another $4n$ memory space.

Thus, the worst-case memory usage of the data structure is $4(k+n)$, whilst properly selected q, h and k parameters pare down the required memory to $4n$.

6 Experimental results

In this section we compare our implementation of the hash table version with other available suffix tree and suffix array implementations. All of the tests ran on a system using Intel Core i5-2400 3.1 GHz CPU and 2 GB available main memory.

6.1 Compared Implementations

The following implementations were compared:

STree: This implementation is based on suffix trees, using Ukkonen's algorithm for construction [13] and written in C. The construction is linear in $|S|$ and matching a pattern with z occurrences takes $O(z + |P|)$ time, thus once the tree is built matching a pattern is independent of the length of S.

DivSuf: This is a suffix array using version. For the construction we use Yuta Mori's C implementation [10], which is deemed to be one of the fastest. However, for matching patterns we simply use binary search, although with two additional auxiliary data structures one could achieve the desired $O(z + |P|)$ matching time [3].

SH_QX_KY: Finally, this is our own C++ based implementation based on a special hash table implemented by us. X and Y denote the actual value of the q and k parameters, for example *SH_Q5_K1.5* means that $q = 5$ and $k = 1.5n$ parameters were used in the tests.

The implementations were tested with a variety of input files ranging from DNA sequences to source codes. A detailed description of the different characteristics of the source files can be found at http://pizzachili.dcc.uchile.cl/ which is widely used for testing string processing algorithms.

6.2 Construction and Pattern Matching

For the first test we used patterns of 30 to 100 length randomly selected from the source file. Table 2 summarizes the construction and pattern matching times of the different implementations. The construction times refer to the time needed to build the data structure, while the matching time denotes the elapsed time for matching a million patterns with the previously built data structure (all times are in milliseconds). The leftmost numbers ranging from 100000 to 3000000 denote the size of the input files in bytes, that is, the length of the original string, in which we would like to search for the patterns.

While the construction times of the suffix array and the hash table are really close to each other, there is a huge discrepancy in pattern matching times between these data structures. Exact string matching with the hash table is especially efficient for varied inputs like DNA, proteins and English text.

Since the suffix array and the hash table based version clearly outperforms the suffix tree based version for larger strings both in construction and in pattern matching times, we did a second evaluation for larger inputs with the suffix array and hash table based versions. Here we used input files with size of 50MB and matched a million patterns in each case. Table 3 summarizes the results of this comparison.

Because we created patterns by randomly reading 30 to 100 characters of the given source file, we matched frequent patterns more often. For example, in the ”pitches” file there are more than 300 000 matches for the 30 long pattern consisting of the * character, which means that the bucket associated with this pattern contains at least 300 000 elements.

In addition to the large bucket sizes, we were matching these kind of patterns many times, because these are the most probable candidates when we select a random pattern from the source. Thus, we had to operate on immensely large buckets for most of the time. In real life application this is usually not a problem, since we do not want to search for the ”30 long *” pattern this often, hence these large buckets can usually be ignored.

This is why we also used another type of pattern generation in the detailed comparison of the suffix array and hash table versions. In the latter pattern generation method, we generated patterns randomly from a given alphabet, which means that matches were fairly unlikely, thus we did not have to operate on large buckets. In Table 3 ”From Source” indicates that the patterns were selected from the source file itself, while ”Random” means that the patterns were randomly generated.

It is important to understand that occasionally dealing with large buckets is not a problem at all, what really slows down the process is the frequent matching of these repeating patterns.

Table 2: Construction and execution times in milliseconds for input files of different size and type.

		Construction				Matching			
		STree	DivSuf	SH_Q20_K1.5	SH_Q30_K4	STree	DivSuf	SH_Q20_K1.5	SH_Q30_K4
100000	dna	41.8	10.9	8.0	**6.7**	577.6	630.2	196.2	**129.1**
	sources	37.9	**8.1**	9.1	9.3	1369.8	653.6	415.0	**286.0**
	proteins	52.6	10.5	**9.5**	10.0	871.2	575.9	468.1	**365.2**
	pitches	53.2	8.4	8.3	**8.0**	1189.7	529.1	203.2	**125.5**
	english	57.4	9.2	8.0	**6.9**	1612.9	609.0	198.7	**124.4**
	dblp.xml	29.5	7.7	7.1	**4.3**	2080.9	739.7	941.0	**669.9**
1000000	dna	447.3	86.6	**73.0**	79.7	1301.7	1006.4	328.7	**200.1**
	sources	355.4	**68.3**	75.8	84.2	4971.7	1058.4	1154.7	**682.9**
	proteins	610.6	**76.6**	77.2	86.5	2133.4	931.5	682.1	**517.6**
	pitches	850.2	**73.7**	75.6	76.1	3719.6	904.9	362.7	**231.8**
	english	566.1	**73.6**	82.9	75.9	4204.6	980.6	388.3	**212.8**
	dblp.xml	306.8	**53.2**	78.1	81.8	10976.7	**1313.8**	4371.6	2556.8
3000000	dna	1625.8	240.2	247.0	**224.1**	1995.4	1584.8	402.7	**225.7**
	sources	1302.2	**180.5**	239.4	240.1	8104.7	1649.2	1228.4	**741.1**
	proteins	2732.0	259.3	**233.1**	241.0	3184.3	1567.4	703.8	**518.7**
	pitches	2794.8	**184.0**	235.3	241.0	7070.9	1490.7	704.5	**509.1**
	english	2230.5	**218.1**	246.3	220.1	6350.0	1496.0	404.7	**231.5**
	dblp.xml	1105.0	**166.2**	215.8	223.1	16317.8	**2026.0**	6920.9	3544.8

Table 3: Construction and execution times in seconds for input files of different type with a size of 50MB.

	Construction			**Matching**					
				From Source			**Random**		
	DivSuf	SH_Q20_K1.5	SH_Q30_K4	DivSuf	SH_Q20_K1.5	SH_Q30_K4	DivSuf	SH_Q20_K1.5	SH_Q30_K4
dna	6.18	5.22	**4.66**	3.30	0.72	**0.36**	0.731	0.163	**0.094**
sources	**4.09**	4.72	4.80	**3.69**	16.62	6.41	1.354	0.135	**0.078**
proteins	7.43	**4.64**	4.65	3.14	1.66	**0.93**	1.933	0.160	**0.085**
pitches	**4.20**	4.51	4.46	**10.62**	103.12	83.67	2.113	0.167	**0.085**
english	6.03	6.38	**5.82**	3.21	1.21	**0.79**	1.006	0.172	**0.104**
dblp.xml	**4.20**	4.49	5.64	**4.89**	61.91	33.50	1.169	**0.073**	0.087

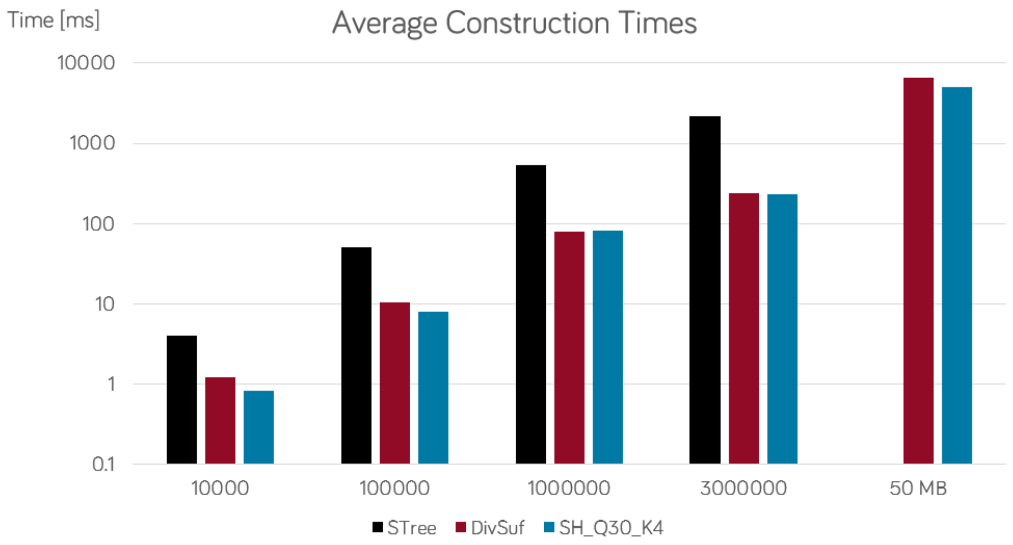

Figure 12: The average construction times for different input sizes. The average was calculated from the DNA, protein and English input files.

Although the construction times of the suffix array and the hash tables are close to each other (see in Figure 12), a suffix array alone is not capable of matching patterns in linear time to their length, which resulted in longer matching times (see in Figure 13). It seems that when there are no occurrences at all (in the case of randomly generated patterns) the hash table is more than ten times faster than the suffix array. The reason for this is that we only have to generate a single hash value and there is a really high chance that the first bucket is already empty and we can terminate the search.

To achieve the desired $O(z+|P|)$ matching time with the suffix array, we have to build two additional data structures, the LCP-array and the child-table [3]. However, even the fastest currently known method for constructing the LCP-array takes about the same time as the suffix array construction [5], let alone the child-table construction. This means that in many cases constructing an enhanced suffix array alone would take three times more time than constructing a hash table and matching a million patterns to it.

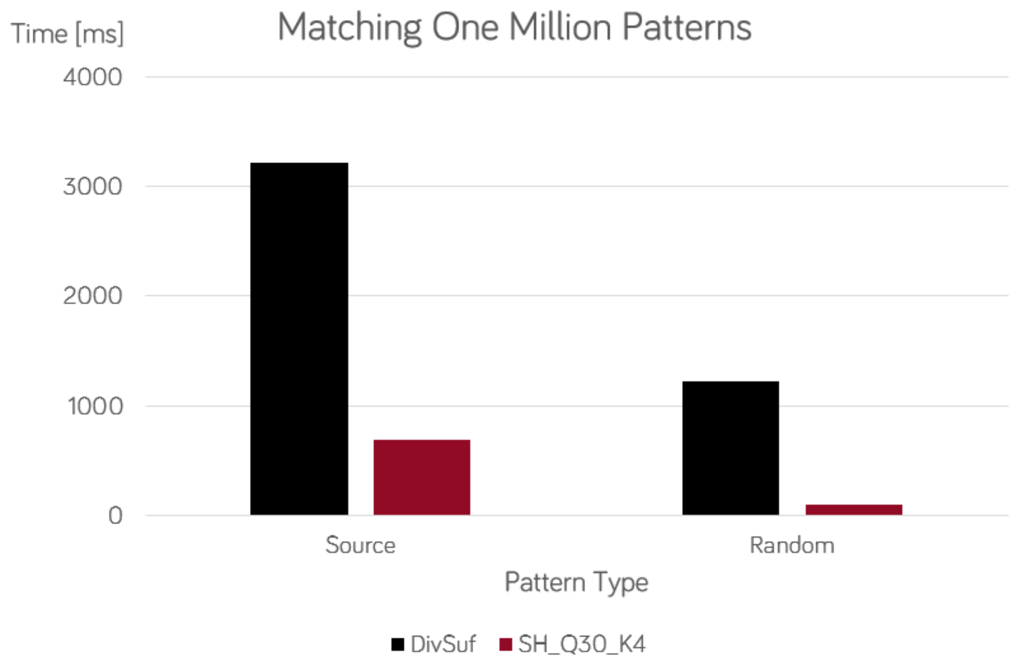

Figure 13: The average matching times of 1 million patterns. The average was calculated from the 50MB DNA, protein and English input files. The source label denotes that the patterns were randomly selected from the source file itself, whereas random denotes that the pattern were randomly generated from an alphabet.

The advantage of our approach is that it provides an efficient method for predictably long pattern strings and diversified S string where there are a small amount of matches on average, while the disadvantage of this technique is that it is only usable efficiently for searching patterns over certain length. Furthermore the algorithm is not prepared for any changes in the string S during the searches, and it is incapable of finding partial matches of the search pattern in its current form. The algorithm is also sensitive to badly-chosen parameters that can drastically slow down the process due to large bucket sizes.

7 Conclusion and Outlook

This paper introduced a novel method for using and implementing hash tables to solve the exact string matching problem. It was also shown by measurements that under certain circumstances it outperforms suffix tree and suffix array based approaches.

The construction time of this data structure is in many cases the same as that of the suffix array's, but it provides a much faster way to match long patterns, while constructing an enhanced suffix array, which is capable of matching patterns in linear time to the length of P, takes much more time then preparing the hash table.

We also showed how to choose the most important parameters k and q to provide a fast way for matching long patterns in various S strings, where the average number of matches remains low. If the fluctuation of the length of P is small or there is no fluctuation at all, this is an especially efficient method.

For future improvement we suggest a well-chosen data structure for buckets, while our future intention is to investigate the parallelizability and improving the cache friendliness of the approach.

References

[1] M.I. Abouelhoda and A. Dawood. Fine tuning the enhanced suffix array. In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–4, Dec 2008.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, March 2004.

[3] MohamedIbrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In AlbertoH.F. Laender and ArlindoL. Oliveira, editors, *String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer Berlin Heidelberg, 2002.

[4] A. Dudas and J. Sandor. Cache performance and efficiency factors of parallel data structures. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 580–587, July 2012.

[5] Johannes Fischer. Inducing the lcp-array. In Frank Dehne, John Iacono, and Jrg-Rdiger Sack, editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer Berlin Heidelberg, 2011.

[6] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997.

[7] Sndor Juhsz and kos Duds. Adapting hash table design to real-life datasets. In *Proc. of the IADIS European Conference on Informatics 2009, part of the IADIS Multiconference of Computer Science and Information systems 2009*, pages 3–10, Algarve, Portugal, June 2009.

[8] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.

[9] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software - Practice and Experience*, 29:1149–1171, 1999.

[10] Yuta Mori. An implementation of the induced sorting algorithm. `https://sites.google.com/site/yuta256/sais/`, 2010. [Online; accessed April, 2014].

[11] Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyr, editors, *String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 51–62. Springer Berlin Heidelberg, 2009.

[12] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2005.

[13] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.