Average Optimal String Matching in Packed Strings*

Djamal Belazzougui[1] and Mathieu Raffinot[2]

[1] Department of Computer Science, University of Helsinki, Finland
djamal.belazzougui@cs.helsinki.fi
[2] LIAFA, Univ. Paris Diderot - Paris 7, 75205 Paris Cedex 13, France
raffinot@liafa.univ-paris-diderot.fr

Abstract. In this paper we are concerned with the basic problem of string pattern matching: preprocess one or multiple fixed strings over alphabet σ so as to be able to efficiently search for all occurrences of the string(s) in a given text T of length n. In our model, we assume that text and patterns are tightly packed so that any single character occupies $\log \sigma$ bits and thus any sequence of k consecutive characters in the text or the pattern occupies exactly $k \log \sigma$ bits. We first show a data structure that requires $O(m)$ words of space (more precisely $O(m \log m)$ bits of space) where m is the total size of the patterns and answers to search queries in average-optimal $O(n/y)$ time where y is the length of the shortest pattern ($y = m$ in case of a single pattern). This first data structure, while optimal in time, still requires $O(m \log m)$ bits of space, which might be too much considering that the patterns occupy only $m \log \sigma$ bits of space. We then show that our data structure can be compressed to only use $O(m \log \sigma)$ bits of space while achieving query time $O(n(\log_\sigma m)^\varepsilon / y)$, with ε any constant such that $0 < \varepsilon < 1$. We finally show two other direct applications: average optimal pattern matching with worst-case guarantees and average optimal pattern matching with k differences. In the meantime we also show a slightly improved worst-case efficient multiple pattern matching algorithm.

1 Introduction

The string matching problem consists of finding all occurrences of a given pattern $p = p_1 p_2 \dots p_m$ in a large text $T = t_1 t_2 \dots t_n$, both sequences of characters from a finite character set Σ of size $\sigma = |\Sigma|$. This problem is fundamental in computer science and has a wide range of applications in text retrieval, symbol manipulation, computational biology, and network security.

This problem has been deeply studied and since the 70's there exist algorithms like the Morris-Pratt algorithm (MP) [27] or the well-known Knuth-Morris-Pratt (KMP -a variation of MP) [24] that are average and worst case time $O(n + m)$, that is, linear in the size of the text and that of the pattern. This problem has been further investigated and a time average lower bound in $\Omega(n \log_\sigma(m)/m)$

* Work partially supported by the french ANR-2010-COSI-004 MAPPI Project.

P.G. Spirakis and M. Serna (Eds.): CIAC 2013, LNCS 7878, pp. 37–48, 2013.

(assuming equiprobability and independence of letters) has been proved by A.C. Yao in [33]. Since this seminal work, many average optimal algorithms have been proposed, from both a theoretical point of view, like the Backward Dawg Matching (BDM) [15], than from a practical point of view, like the Backward Nondeterministic Dawg Matching (BNDM) [29]. Worst case linear and average optimal algorithms also appeared, mainly by combining a forward algorithm like KMP with a backward search algorithm similar to BDM or BNDM. The Double-Forward [2] is however an exception to this general approach since it combines two forward algorithms sharing a unique data structure. It is the simplest algorithm reaching these optimal complexities published so far.

A natural extension to the string matching problem is to search for all occurrences of multiple strings instead of a single one. We extend the notation m to be the total size of the patterns. In a similar way to the single pattern case, there exist $O(n + m)$ linear time algorithms, the most famous being the Aho-Corasick algorithm [1], and also average optimal algorithms like Multi-BDM [29] and Dawg-Match [14].

The optimal results and algorithms we mentioned above are valid in classical models. In this paper we are concerned with the single and multiple string pattern matching problem in the RAM model with word length $\omega = \Omega(\log(n+m))$, assuming that text and patterns are tightly packed so that any single character occupies $\log \sigma$ bits and thus any sequence of k consecutive characters in the text or the pattern occupies exactly $k \log \sigma$ bits.

The single and multiple string matching problems have already been studied in this model both from a worst case and average point of view. For the worst case bound, the main studies are from Fredriksson [18], Bille [9] and Belazzougui [3,4]. Those studies made some progress on both single and multiple string matching problems. Very recently Benkiki et al. [8] obtained the optimal $O(n\frac{\log \sigma}{\omega} + occ)$ query time for the single string case, however their result requires the availability of non-standard instructions. Eventually Breslauer et al. [10] obtained an algorithm with the unconditional optimal $O(n\frac{\log \sigma}{\omega} + occ)$ query time for the single string matching problem. The optimal worst-case bound for the multiple string variant is still an open problem despite the progress made in the recent studies.

This paper however mainly focuses on average optimal string matching. This question has already been considered by Fredriksson also in [18] where he presented a general speed-up framework based on the notion of super-alphabet and on the use of tabulation (four russian technique). The single string matching algorithm obtained using these techniques is optimal on average, but at the cost of a huge memory requirement. Precisely, the algorithm is $O(n/y)$ where y is the length of the shortest pattern ($y = m$ in case of a single pattern) while requiring $O(\sigma m)$ space.

In this paper we first explain a data structure that requires only $O(m)$ words of space and answers to queries in average optimal $O(n/y)$. However, our first data structure, while leading to optimal query times, still requires $O(m \log m)$ bits of space which might be still too much considering that the patterns occupy only $m \log \sigma$ bits of space. We then show that our data structure can be

compressed to only require $O(m \log \sigma)$ bits of space while achieving query time $O(n(\log_\sigma m)^\varepsilon / y)$. We eventually show two other direct applications: average optimal pattern matching with worst case guarantees and average optimal string pattern matching with k differences. In the meantime we also show improved solutions for worst-case efficient multiple pattern matching algorithm (which we use for the average-optimal pattern matching with worst-case guarantees).

Model and Assumptions. In this paper, we assume a standard word RAM model with word length $\omega = \Omega(\log n)$ (where n is the size of the input) with all standard operations (including multiplication and addition) supported in constant time. We assume that the text and the patterns are from the same alphabet σ. In our bounds we often refer to three different kinds of time measures:

1. Worst-case time of an algorithm refers to a deterministic time that holds regardless of the input and of the random choices made by the algorithm.
2. Randomized or randomized expected time of an algorithm measures the average time of the algorithm over all possible random choices of the algorithm regardless of the input (it holds for any chosen input).
3. Expected or average time of an algorithm measures the average time of the algorithm considering a probability model on the input. In our case the input is either a text or a pattern and the probability model simply assumes that the positions of the input are all independent and of the same probability $1/\sigma$.

We quantify the space either in bits or in words. To translate between words and bits, the space is simply multiplied by a factor $\log n$. This is justified, since the model only assumes that $\omega = \Omega(\log n)$ and thus allows the case $\omega = \Theta(\log n)$. Note also that even if $\omega \gg \log n$, we can still simulate any algorithm designed to use words of size $\Theta(\log n)$ instead of ω with only a constant-factor slowdown.

2 The Basic Algorithm

We first consider the single string matching problem and we state the following result.

Theorem 1. *Given a pattern (string) p of length m over an alphabet of size σ, we can build a data structure of size $O(m \log m)$ bits so that we can report all the occ occurrences of the pattern p in any packed text T of length n in expected time $O(n/m)$. The construction of the data structure takes $O(m)$ randomized expected time (in which case the data structure occupies $m \log m + O(m)$ bits only) and $O(m \log \log m)$ time in the worst-case.*

Proof. We first present the randomized construction. We use a window of size m characters. We build a minimal perfect hash function f [21] on the set F_t of factors of the pattern of length $t = 3 \log_\sigma m$ characters. This hash function occupies $O(|F_t|)$ bits of space and can be evaluated in $O(1)$ time. Before building this minimal perfect hash function we first extract the factors in the following

way: we traverse the pattern p and then at each iteration i extract in $O(1)$ time the characters $s_i = p[i..i+t-1]$ of the pattern (note that s_i is actually just a bitvector of $3 \log n$ bits) and add the pair (s_i, i) at the end of a temporary array M initially empty. Then, we just sort this array (using radix-sort) in time $O(m)$ where the pairs are ordered according to their first component (the s_i component). Then we store a table $T[1..m]$. For every $s \in F_t$ occurring at position j in p (that is we have $s = p[j..t-1]$), we set $T[f(s)] = j$. Thus the hash function f can be used in conjunction with the table T in order to get at least the position of one occurrence of every substring of p of length t. Note that the space occupancy of f is $O(m-t) = O(m)$ bits while the space occupancy of T is $(m-t)\log m \leq m \log m$ bits. Note also the hash function f can be evaluated in $O(1)$ time as it operates on strings of length $t = 3 \log_\sigma m$ characters, which is $O(\log m) = O(w)$ bits. Next, the matching of p in a text T follows the usual strategy used in the other average optimal string matching algorithms. That is, we use a window (a window is defined by just a pointer to the text plus an integer defining the size of the window in number of characters) of size $2m - t$ (except at the last step where the window could be of smaller size). Before starting the first iteration, the window is placed at the beginning of the text (that is, take $w = T[1..2m-t]$). Then at each iteration i starting from $i = 1$ do the following:

1. Consider the substring $q = w[m-t+1..m]$.
2. Compute $j = T[f(q)]$ and compare q with the substring $p[j..j+t-1]$.
3. In case the two substrings match, do an intensive search for p in the window using any string matching method that runs in reasonable time.
4. Check whether $i(m-t+1)+m \geq n$ in which case the search is finished, otherwise continue with the next step.
5. Finally shift the window by $m-t+1$ positions by setting $w = T[i(m-t+1)+1..(i+1)(m-t+1)+m]$ (or $w = T[i(m-t+1)+1..n]$ whenever $(i+1)(m-t+1)+m > n$), increment i and go to step 1 of the next iteration.

It is easy to check that the algorithm above runs in expected time $O(n/m)$ time assuming that the characters are generated independently uniformly at random. The query time follows from the fact that we are doing $O(n/(m-t+1)) = O(n/m)$ iterations and at each iteration the only non-constant step is the intensive search, which costs $O(m^2)$ (assuming the most naive search algorithm) but takes place only with probability at most $O(m/\sigma^{3\log_\sigma m}) = (1/m^2)$ and thus only contributes $O(1)$ to the cost of search.

We now describe the deterministic construction. Instead of storing a perfect hash function on all factors of lengths $t = 3 \log_\sigma m$ characters, mapping them to interval $[1..m]$ and then storing their corresponding pointers in the table T, we instead store all those factors using the deterministic dictionary of [30]. This dictionary occupies linear space and can be constructed in time $O(n \log\log n)$ when built on a set of n keys of lengths $O(\log n)$ bits each. In our case, we have $\Theta(m)$ keys of length $t = 3 \log_\sigma m$ characters, which is $3 \log m$ bits and thus the construction of [30] takes $O(m \log\log m)$ worst-case time. Note that the space occupancy of this construction is also linear $O(m \log m)$ bits of space and the query time is constant.

What remains is to show how to combine the two data structures. The combination consists only in first trying to build the randomized data structure fixing some maximal amount of time cm for a suitably chosen c. Then if the construction fails to terminate in that time, we build the deterministic data structure. Note, however that the deterministic construction will be very far from practical both from the space and time point of view. □

This result improves upon Fredriksson's result since the memory our algorithm requires is only $O(m)$ words of space or $O(m \log m)$ bits. Using the same approach we extend our result to optimally solve in average the multiple string matching problem.

Theorem 2. *Given a set of patterns (strings) S of total lengths m over an integer alphabet of size σ where the shortest pattern is of length $y \geq 4 \log_\sigma m$, we can build a data structure of size $O(m \log m)$ bits so that we can report the occurrences of any of the patterns in S in any packed text T of length n in expected time $O(n/y)$. The construction of the data structure takes $O(m)$ randomized expected time (in which case the data structure occupies $m \log m + O(m)$ bits only) and $O(m \log \log m)$ time in the worst-case (in which case the data structures occupies $O(m \log m)$ bits of space).*

Proof. The algorithm is almost the same as that of theorem 1 except for the following points:

1. The window size is now $2y - t$ where y is the length of the shortest pattern in the set of patterns and $t = 3 \log_\sigma m$.
2. We index all the substrings of length $t = 3 \log_\sigma m$ of all of the patterns. That is, the function f will store all factors of the strings.
3. The intensive search looks for all of the patterns in the window. The time for this intensive search is $O(ym) = O(m^2)$.
4. The window is advanced by $y - t + 1$ characters at each iteration.

The query time can be easily bounded by the same analysis used in theorem 1. That is, at each iteration the only non-constant time step is the intensive search that takes $O(ym) = O(m^2)$ time, but is triggered only with probability $O(1/m^2)$ and thus contributes a $O(1)$ cost. The probability $O(1/m^2)$ is deduced from the fact that any string of length $3 \log_\sigma m$ generated at random has a probability $O(1/\sigma^{3 \log_\sigma m}) = O(1/m^3)$ of colliding with any substring of one of the patterns and thus, the probability of colliding with any substring of any of the patterns is $O(1/m^2)$. Note that the window is advanced by $t' = y - t + 1 = y - 3 \log_\sigma m + 1$ at each iteration. Since $y \geq 4 \log_\sigma m$, we deduce that $t' \geq y/4$. Thus we have about $4n/y$ iteration where at iteration an expected $O(1)$ time is spent which gives a total of $O(n/y)$ time. □

3 Succinct Representation

The space required by the representation used in the two previous theorems is $O(m \log m)$ (only $m \log m + O(m)$ bits for a randomized construction).

This space usage is not succinct in the sense that it is larger than the space used by the pattern(s), which is $m \log \sigma$ bits only. In the context of single string matching, this is probably not a problem as m is small enough to fit entirely in memory. However in the case of multiple string matching, the cumulative size of the patterns may exceed the available memory (or at least do not fit into fastest levels of memory). In the last decade many efforts have been devoted to reduce the space required by the full text indexing data structures that allow us to answer efficiently (with a small sacrifice in query time) to queries asking for all the occurrences of a pattern in a text, but in which the text is fixed and the patterns are given as queries. Note that this is the converse of our case, in which the text is given as a query and the pattern(s) is/are fixed. In this section we show that the first two theorems can also be modified to obtain a different space/time trade-off between the space used by the data structure and the time required to match the text. Our solution makes use of results from succinct full-text indexing. our solution makes use of results from succinct full-text indexing literature, namely the compressed text index of [19] built using the space-efficient methods described in [22,23]. In [19] the following lemma was shown:

Lemma 1. *There exists a succinct full text index which can be built on a text T of length m (or a collection of texts of total length m) over an alphabet of size σ such that:*

1. *The full text index occupies space $O(m \log \sigma)$ bits and;*
2. *It can return for any string p the range of suffixes (a range $[i..j]$ in the suffix array) of T prefixed by p in time $O(|p|/\log_\sigma m + (\log_\sigma m)^\varepsilon)$.*

The construction of the data structure takes either $O(m \log \sigma)$ randomized time or $O(m \log m)$ deterministic time and uses $O(m \log m)$ bits of space during the construction.

The construction algorithm in [19] uses $O(m \log m)$ bits of temporary space during the construction though the constructed index occupies only $O(m \log \sigma)$ bits of space. In our case we strive to reduce the peak consumption during both construction and matching phases and thus need to use a construction algorithm that uses space comparable to the constructed index. This is achieved by the following result:

Lemma 2 ([22,23]). *Given a text T of length n (or a collection of texts of total length m) over an alphabet of size σ, the indexes of [19] can be constructed in worst-case time $O(m \log m)$ using temporary space $O(m \log \sigma)$ bits of space (in addition to the final space of the index).*

The result about our succinct representation for average-optimal string matching is summarized by the following theorem:

Theorem 3. *Given a set of patterns (strings) S of total lengths m over an integer alphabet of size σ where the shortest pattern is of length $y \geq 4 \log_\sigma m$, we can build a data structure of size $O(m \log \sigma)$ bits so that we can report all of the occurrences of any of the patterns in time $O(n(\log_\sigma m)^\varepsilon / y)$ for any $\varepsilon \in (0, 1)$.*

The index can be built in worst-case time $O(m \log m)$ using $O(m \log \sigma)$ bits of temporary space.

Proof. For achieving the improved space bounds we will incorporate lemma 1 into the first step of the algorithm, which was to find whether the string $q = w[m - t + 1..m]$ matches some substring of some text. The hash data structure used for that purpose in theorem 2 occupies space $O(m \log m)$. The reason for needing that much space was because of the table T which stores m pointers of $\log m$ bits each. We now show an alternative way to match the strings using a compressed index. For that, we simply index the pattern(s) using the index of lemma 1. Then, checking if q exists takes $O(t/\log_\sigma m + (\log_\sigma m)^\varepsilon) = O((\log_\sigma m)^\varepsilon)$ time. The space used by the index is $O(m \log \sigma)$ bits. Building the index (using lemma 2) takes $O(m \log m)$ time and uses at most $O(m \log \sigma)$ bits of temporary space. □

The main advantage of theorem 3 over the first two theorems is that the peak memory use during the preprocessing or the matching phases never exceeds $O(m \log \sigma)$ bits of space (against $\Theta(m \log m)$ bits of peak memory use in the two first theorems). The drawback is that the construction and matching phases are slightly slower than those of theorems 2 and 1.

Corollary 1. *Given a pattern p of length m over an integer alphabet of size σ, we can build in time $O(m \log m)$ a data structure of size $O(m \log \sigma)$ bits so that we can report all the occ occurrences of p in any packed text T of length n in expected time $O(\frac{n}{m}(\log_\sigma m)^\varepsilon)$ for any $\varepsilon \in (0, 1)$. The peak memory usage during the construction or matching phases is $O(m \log \sigma)$ bits.*

Note that this corollary still improves when compared with the standard algorithm which examines the pattern and the text character by character and has query time $O(\frac{n}{m} \log_\sigma m)$, slower than our corollary by a factor $(\log_\sigma m)^{1-\varepsilon}$.

4 Worst-Case Guarantees

The optimal average time algorithm we presented above runs in worst case time $O(\frac{n}{m} + occ)$. However, in many real life uses of string matching, one does not know any property of the data source. Thus, even if on average the algorithm is optimal, it is cautious to bound its worst case time to avoid a *bad* instance to block the software. We thus extend our approach to guarantee a worst case time bound at least as worst as the fastest forward algorithm designed for (multiple) string(s) matching when text and patterns are tightly packed. For this sake we adapt the approach used in the Double-Forward algorithm [2] to packed strings.

The idea is simple. We consider the same approach used for proving Theorem 1. We hash the last $3 \log_\sigma m$ characters of the current search window through the minimal perfect hashing function we built on all $3 \log_\sigma m$ characters long factor of the pattern. If the hash test fails, we shift the search window to the right just after the first character of the $3 \log_\sigma m$ last characters we tested. This situation is shown in figure 1. Note that this step is the same as in the proof of Theorem 1.

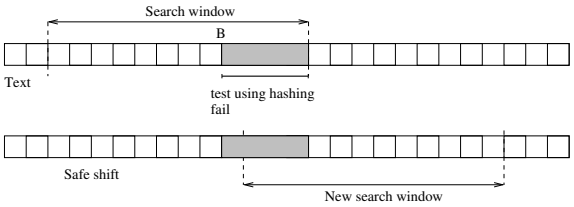

Fig. 1. Hash test fail case. The search window is shifted after the first of the $3 \log_\sigma m$ characters tested.

The algorithm changes if the test passes. Assume that at some point we began a forward scan of the text using the algorithm of [4] that stopped in a position A in the text after having read a prefix u of the current search window. Figure 2 illustrates this case. We simply continue this forward search from A', passing the end of the current search window and reading characters until the longest suffix of the text (read by blocks of characters) that matches a prefix u' of the pattern is small enough. We then repeat the global search scheme from this new window.

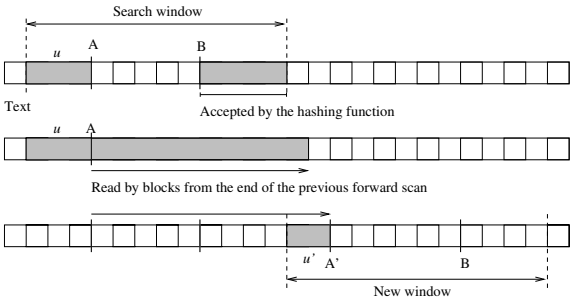

Fig. 2. Hash test success case. The forward search stopped in A' is continued until passing the end of the initial window until point A'.

It is obvious that this new algorithm remains optimal on average. As a forward algorithm, we use the very recent algorithm of Breslauer et al. [10]. The algorithm achieves preprocessing time $O(m/\log_\sigma m+\omega)$ and optimal worst-case query time $O(n\frac{\log\sigma}{\omega})$. We thus get the following result:

Theorem 4. *Given a pattern p of length m characters over an integer alphabet of size σ, we can build in randomized $O(m+\omega)$ time and worst-case $O(m\log\log m+\omega)$ time a data structure of size $O(m\log m)$ bits so that we can report all of the occ occurrences of p in any packed text T of length n in expected time $O(n/m)$ and worst-case time $O(\frac{n\log\sigma}{\omega}+occ)$.*

4.1 Multiple String Matching

We extend the previous algorithm to match a set of strings. We use as a forward algorithm an improved version of the multiple string algorithm searching

described in [3,4], whose original version runs in $O(n(\frac{\log d+\log y+\log\log m}{y}+\frac{\log\sigma}{\omega})+occ)$ time using a linear-space data structure that can be built in $O(m\log m)$ time. We first prove the following (slightly) improved result based on [3,4]:

Theorem 5. *Given a set of d patterns (strings) S of total length m characters over an integer alphabet of size σ where the shortest pattern is of length y, we can build in worst case time $O(m\log m)$ a data structure of size $O(m\log m+\omega)$ bits so that we can report all of the occ occurrences of any of the patterns in S in any packed text T of length n in worst-case time $O(n(\frac{\frac{\log d}{\log\log d}+\log y+\log\log m}{y}+\frac{\log\sigma}{\omega})+occ)$. The data structure uses temporary space $O(m\log m+\omega^2)$.*

Proof. To get the improved result we first notice that the text matching phase in the multiple string matching algorithm of [4] proceeds in $\Theta(n/y)$ iterations, at each iteration reading y consecutive characters and spending $O(\log d+\log y+\log\log m+y\frac{\log\sigma}{\omega})$ time, where the two terms $\log d$ and $\log y$ are due to :

1. The use of a string B-tree, which is a data structure used for longest suffix matching. The string B-tree stores a set S' of $O(dy)$ strings (actually $O(dy)$ pointers to text substrings of length y) and answers to a longest suffix matching query for a query string of length y in time $O(y\frac{\log\sigma}{\omega}+\log(dy))$.
2. The use of a two dimensional rectangle stabbing data structure of Chazelle [13]. This data structures stores up to $m'=O(dy)$ rectangles using $O(m')=O(dy)$ space and answers to rectangle stabbing queries that ask to report all the rectangles that contain a given query point. The original query time was $O(\log m'+occ)$ where occ is the number of reported rectangles. However this query time was later improved to $O(\frac{\log m'}{\log\log n'}+occ)$ [34].

To get our improved query time, we will use an alternative data structure for longest suffix matching. We use a compacted trie built on the set S'. The compacted trie will use $O(dy)$ pointers of $O(\log(dy))=O(\log m)$ bits. The compacted trie can be built in time $O(m\log m)$ and uses space $O(m\log m)$ bits. In order to accelerate the traversal of the trie we will use super-characters built from every $\frac{\omega}{\log\sigma}$ consecutive characters. The compacted trie will be built on the original set of strings considered as strings over the super-alphabet. This is done by grouping every sequence of $\frac{\omega}{\log\sigma}$ characters into a single one (strings whose length is not multiple of $\frac{\omega}{\log\sigma}$ are padded with special characters before grouping). Then a node of the trie will have children labeled with characters from the new alphabet. Note that each character of the new alphabet occupies $O(\omega)$ bits. In order to be able to determine which child to follow, we need to build a dictionary on the set of labels of each node. However this would occupy in total $O(m\omega)$ bits of space. In order to reduce the space, we will group all the distinct labels of all the nodes in the trie, and build a perfect hash function that maps all the distinct labels to a range of size $m^{O(1)}$ (we call the resulting labels reduced labels). The resulting function occupies $O(\omega)$ bits of space . A deterministic perfect hash function occupying constant space can be built in deterministic $O(m\log m)$ time [20] or randomized $O(m)$ time using universal hashing [11]. Then for each

node we build a local deterministic dictionary that associates a pointer to the child corresponding to each reduced label (we call it child dictionary). As each reduced label is the range $m^{O(1)}$ means that it can be described using $O(\log m)$ bits. Thus the total space occupied by all child dictionaries will be $O(m \log m)$ bits. The compacted trie above is able to to return the length of the longest suffix matching up to lengths multiple of $\frac{\omega}{\log \sigma}$ (old alphabet) characters.

In order to terminate the longest suffix matching query up to additional $\frac{\omega}{\log \sigma} - 1$ characters, we will for each node build a z-fast trie [5,7] on all child labels. The z-fast trie will be built on the set of labels of the children of each node. A z-fast trie supports longest suffix matching queries in time $O(\log |p|)$ on a pattern p. It uses a linear number of pointers to the original strings (see [7]). The probabilistic z-fast trie presented in [5,7], can be made deterministic by building a deterministic perfect hash function on the path labelling each node in the z-fast trie using [20]. The paths are of length at most ω bits, we can reuse the same strategy above to map the labelling paths to $O(\log m)$ bits.

In order to further reduce the used space we partition our initial set of strings into groups of ω strings and store only the first string of each group in the structure above (we call it sampled trie), but in addition for each node of the sampled trie store a pointer to the corresponding node in the original non-sampled trie. Then we build the same above trie structure on each group of ω (call them group tries). This ensures that building the above structure on each group requires just $O(\omega^2)$ bits of space. That way the total space is bounded by $O(\omega^2 + m \log m)$ bits of space. Finally a query will first start in the sampled trie, then follow a pointer to the non-sampled trie to match at most one additional character. This isolates a range that spans at most two groups which can then be searched using the local group tries. □

We can now replace the $\log d + \log y$ term with the addition of the query times of the alternative longest suffix matching data structure and the improved query time for the rectangle stabbing problem to obtain a total query time $O(\frac{\frac{\log(dy)}{\log\log(dy)} + \log y + \log\log m}{y} + occ) = O(\frac{\frac{\log d}{\log\log d} + \log y + \log\log m}{y} + occ)$.

Now back to the average-optimal multiple string matching algorithm. As the size of the search window in the algorithm is y, the size of the smallest string searched, the complexities we obtain on average depend on y instead of m. And we get:

Theorem 6. *Given a set of d patterns (strings) S of total length m characters over an integer alphabet of size σ where the shortest pattern is of length $y \geq 4 \log_\sigma m$, we can build in worst case time $O(m \log m)$ a data structure of size $O(m \log m)$ bits so that we can report all of the occ occurrences of any of the patterns in S in any packed text T of length n in expected time $O(n/y)$ and in worst case time $O(n(\frac{\frac{\log d}{\log\log d} + \log y + \log\log m}{y} + \frac{\log \sigma}{\omega}) + occ)$.*

5 Approximate String Matching

Approximate string matching considering edit distance is to find all positions in a text where at least one string of the set of patterns matches the text up to k errors. The approximate string matching problem can be efficiently solved for relatively small k (in regard to the length of the strings) using exact string matching as a filter. This approach is also used in [18]. The idea is the following. Assume a string to match the text up to k errors. If this string is first divided into $k+1$ pieces, one of these pieces must exactly match the text. Thus, the approach is to split the string patterns in $k+1$ pieces and search for all those pieces simultaneously in the text using the multiple string matching algorithm of Theorem 2. If the pieces are of the same length, $m/(k+1)$, the average time required to search all pieces is $O(n/(m/k+1)) = O((k+1)n/m) = O(kn/m)$.

If one of those pieces matches, then a complete match of the corresponding strings is checked using a classical $O(km)$ algorithm.

Considering a probability model in which all positions are independent and of the same probability $1/\sigma$, a rough upper bound of the number of verifications is $O(nk(1/\sigma)^{(m/k)})$. For $k < m/\log_\sigma m$, the time of the multiple string matching algorithm dominates and the average time remains $O(kn/m)$.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Allauzen, C., Raffinot, M.: Simple optimal string matching. Journal of Algorithms 36, 102–116 (2000)
3. Belazzougui, D.: Worst case efficient single and multiple string matching in the ram model. In: Iliopoulos, C.S., Smyth, W.F. (eds.) IWOCA 2010. LNCS, vol. 6460, pp. 90–102. Springer, Heidelberg (2011)
4. Belazzougui, D.: Worst-case efficient single and multiple string matching on packed texts in the word-ram model. J. Discrete Algorithms 14, 91–106 (2012)
5. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In: SODA, pp. 785–794 (2009)
6. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Fast prefix search in little space, with applications. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 427–438. Springer, Heidelberg (2010)
7. Belazzougui, D., Boldi, P., Vigna, S.: Dynamic Z-fast tries. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 159–172. Springer, Heidelberg (2010)
8. Ben-Kiki, O., Bille, P., Breslauer, D., Gasieniec, L., Grossi, R., Weimann, O.: Optimal Packed String Matching. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011). Leibniz International Proceedings in Informatics (LIPIcs), vol. 13, pp. 423–432 (2011)
9. Bille, P.: Fast searching in packed strings. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 116–126. Springer, Heidelberg (2009)
10. Breslauer, D., Gąsieniec, L., Grossi, R.: Constant-time word-size string matching. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 83–96. Springer, Heidelberg (2012)
11. Carter, L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: STOC, pp. 106–112 (1977)

12. Chan, T.M., Larsen, K.G., Patrascu, M.: Orthogonal range searching on the ram, revisited. In: Symposium on Computational Geometry, pp. 1–10 (2011)
13. Chazelle, B.: Filtering search: A new approach to query-answering. SIAM J. Comput. 15(3), 703–724 (1986)
14. Crochemore, M., Czumaj, A., Sieniec, L.G., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Fast multi-pattern matching. Rapport I.G.M. 93-3, Université de Marne-la-Vallée (1993)
15. Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press (1994)
16. Czumaj, A., Crochemore, M., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., Rytter, W.: Speeding up two string-matching algorithms. Algorithmica 12, 247–267 (1994)
17. Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS, pp. 137–143 (1997)
18. Fredriksson, K.: Faster string matching with super-alphabets. In: Laender, A.H.F., Oliveira, A.L. (eds.) SPIRE 2002. LNCS, vol. 2476, pp. 44–57. Springer, Heidelberg (2002)
19. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
20. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. Algorithms 41(1), 69–85 (2001)
21. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 317–326. Springer, Heidelberg (2001)
22. Hon, W.-K., Lam, T.W., Sadakane, K., Sung, W.-K., Yiu, S.-M.: A space and time efficient algorithm for constructing compressed suffix arrays. Algorithmica 48(1), 23–36 (2007)
23. Hon, W.-K., Sadakane, K., Sung, W.-K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. 38(6), 2162–2178 (2009)
24. Knuth, D.E., Morris Jr, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(1), 323–350 (1977)
25. Lecroq, T.: Recherches de mot. Thèse de doctorat, Université d'Orléans, France (1992)
26. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
27. Morris Jr., J.H., Pratt, V.R.: A linear pattern-matching algorithm. Report 40, University of California, Berkeley (1970)
28. Morrison, D.R.: Patricia-practical algorithm to retrieve information coded in alphanumeric. J. ACM 15(4), 514–534 (1968)
29. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. ACM Journal of Experimental Algorithmics 5, 4 (2000)
30. Ružić, M.: Constructing efficient dictionaries in close to sorting time. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 84–95. Springer, Heidelberg (2008)
31. Ruzic, M.: Making deterministic signatures quickly. ACM Transactions on Algorithms 5(3) (2009)
32. Sadakane, K.: Compressed suffix trees with full functionality. Theory Comput. Syst. 41(4), 589–607 (2007)
33. Yao, A.C.: The complexity of pattern matching for a random string. SIAM J. Comput. 8(3), 368–387 (1979)
34. Shi, Q., JáJá, J.: Novel Transformation Techniques Using Q-Heaps with Applications to Computational Geometry. SIAM J. Comput. 34(6), 1474–1492 (2005)