2009 50th Annual IEEE Symposium on Foundations of Computer Science

Exact And Approximate Pattern Matching In The Streaming Model

Benny Porat
Bar-Ilan University
bennyporat@gmail.com

Ely Porat
Bar-Ilan University
poratel y@cs.biu.ac.il

Abstract— We present a fully online randomized algorithm for the classical pattern matching problem that uses merely $O(\log m)$ space[1], breaking the $O(m)$ barrier that held for this problem for a long time. Our method can be used as a tool in many practical applications, including monitoring Internet traffic and firewall applications. In our online model we first receive the pattern P of size m and preprocess it. After the preprocessing phase, the characters of the text T of size n arrive one at a time in an online fashion. For each index of the text input we indicate whether the pattern matches the text at that location index or not. Clearly, for index i, an indication can only be given once all characters from index i till index $i + m - 1$ have arrived. Our goal is to provide such answers while using minimal space, and while spending as little time as possible on each character (time and space which are in $O(poly \log(n))$).

We present an algorithm whereby both false positive and false negative answers are allowed with probability of at most $\frac{1}{n^3}$. Thus, overall, the correct answer for all positions is returned with a probability of $\frac{1}{n^2}$. The time which our algorithm spends on each input character is bounded by $O(\log m)$, and the space complexity is $O(\log m)$ words.

We also present a solution in the same model for the pattern matching with k mismatches problem. In this problem, a *match* means allowing up to k symbol mismatches between the pattern and the subtext beginning at index i. We provide an algorithm in which the time spent on each character is bounded by $O(k^2 poly(\log m))$, and the space complexity is $O(k^3 poly(\log m))$ words.

1. Introduction

The pattern matching problem is a prominent problem in computer science, and involves finding all the instances of a *pattern* string P, of length m, as a subword (contiguous substring) in *text* string T, of length n (we assume $m < n$). This problem , as well as many variations of it, has been extensively investigated in the past. The naive algorithm tries match every index of the text to the pattern. This algorithm runs in $O(nm)$ time. Knuth, Morris, and Pratt [13], and Boyer and Moore [3] have designed algorithms that require only linear time (proportional to the text) for solving this problem. KMP algorithm can be easily adapted to the online model.

In addition to time complexity, space is another important aspect of problem solutions. Both Knuth, Morris, and Pratt [13], and Boyer and Moore [3] algorithms require at least $O(m)$ space. Galil and Seiferas [11] suggested the first time-space optimal algorithm for the pattern matching problem. Their algorithm works in linear time in relation to the size of the input, and requires only a constant-size memory space in addition to the space occupied by the pattern and text. Following their paper a lot of effort has been devoted to that subject [6], [5], [7], [17], [10].

In [4] we have presented a black box algorithm that can convert most of the offline pattern matching algorithm to the online model, however this conversion required $O(m)$ space. We also proved that a deterministic online pattern matching algorithm **must** use at least $O(m)$ space. This means that in order to break the $O(m)$ space lower bound, we must randomize. Rabin and Karp [15] presented a randomized on-line algorithm that solves the pattern matching problem in linear time. They use fingerprints in order to check if there is a match at a given position in constant time. However, in order to maintain the fingerprint, they needed to save the last m character of the text.

Fast approximate string matching is a central problem of modern data intensive applications. Its applications are many and varied, from computational biology and large scale web searching to searching multimedia databases and digital libraries. As a result, string matching has to continuously adapt itself to the problem at hand. Simultaneously, the need for asymptotically fast algorithms grows every year with the explosion of data available in digital form.

A great deal of progress has been made in finding fast algorithms for a variety of important forms of approximate matching. One of the most studied is the Hamming distance, which measures the number of mismatches between two strings. Given a text t of length n and a pattern p of length m, the task is to report the Hamming distance at every possible alignment. $O(n\sqrt{m \log m})$ time solutions based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [1], [14]. Particular interest has been paid to a bounded version of this problem called the k-mismatch problem. Here a bound k is given and we need only report the Hamming distance if it is less than or equal to k. If the number of mismatches is greater than the bound, the algorithm need only report that fact and not give the actual Hamming distance. In 1985 Landau and Vishkin suggested a beautiful $O(nk)$ algorithm that is not FFT based which uses constant time LCA operations on the suffix tree of p and

[1]This work was supported by BSF and ISF

t [16]. This was subsequently improved to $O(n\sqrt{k \log k})$ time by a method based on filtering and FFTs again [2]. Approximations within a multiplicative factor of $(1+\epsilon)$ to the Hamming distance can also be found in $O(n/\epsilon^2 \log m)$ time [12].

In this paper we suggest a randomized online algorithm for solving the pattern matching problem. Our method uses only $O(\log m)$ space (we neither save the pattern nor the text), and the running time for each character is bounded by $O(\log m)$, however in the average case we spend just a constant time for each character. In addition we present another randomized algorithm for solving the k-mismatch problem. The running time of our algorithm is bounded by $O(k^2 poly(\log m))$ for each character, and the space complexity is bounded by $O(k^3 poly(\log m))$.

Solving the online pattern matching problem with optimal space is very interesting from a theoretical point of view. For a long time it seemed impossible to solve the problem with less than $O(m)$ space even for randomized algorithms. In addition, the problem has many applications in several areas. An example can be taken from the world of intelligence. When the amount of information is tremendous it quickly becomes impossible to save all the information and then search it. Thus, having the ability to search online for patterns in the vast amounts of information, and save only the information where the specific pattern occurred, has great advantages. Other examples are applications that need to monitor Internet traffic, or firewall applications that need to monitor all the information that goes through a specific computer and to block the dangerous information, such as viruses and maleware connections. Another area that can use our method is robotics. When a robot obtains data from its sensors every nanosecond and must immediately react, only limited computing time and space can be used. Several other motivations for our method include automatic stock market analysis and computational biology.

Although the space complexity improvement seems minor, it's very significant. Usually the patterns can not fit inside the fast cache memory. Thus, for every arriving character the algorithm will need to access the RAM in order to compere it to the pattern. RAM accesses are relatively expensive and cause a great slowdown of the algorithm.

The structure of this paper is as follows: we start with some definitions of fingerprint in section 2. Some related work is presented in section 3. We continue with a high level description of our method in section 4. In section 5 our final algorithm for the classical pattern matching problem is present. We present in section 6 a solution for the pattern matching problem with exactly 1-mismatch, by using the algorithm form section 5. We conclude with an algorithm for solving the pattern matching problem with k-mismatches in section 7.

2. Fingerprints

We define the concept of *fingerprint*, one of the basic components of our suggested method.

Definition 1: A fingerprint of a string S is a small string $\phi(S)$ with the following properties:

1) ϕ is a function of S. In particular, if two strings are equal, then so are their fingerprints.
2) $P_r(\phi(S_1) = \phi(S_2)) << 1$ for random strings $S_1 \neq S_2$

Example 1: Let $S = s_1, s_2, ...s_l$ be a string, one can define the fingerprint $\phi_{r_1...r_l}$ of S to be $\phi_{r_1...r_l,p}(S) = (\sum_{i=1}^{l} s_i r_i) \bmod p$ for random $r_1, ...r_l \in \mathbf{F_p}$ and a prime p.

Definition 2: We define *sliding fingerprint* to be a fingerprint that satisfies the two following conditions:

1) It is possible to compute the fingerprint of $s_1, s_2, ...s_l, s_{l+1}$, $(\phi(s_1, s_2, ...s_l, s_{l+1}))$ from the fingerprint of $s_1, s_2, ...s_l$, $(\phi(s_1, s_2, ...s_l))$ and s_{l+1}.
2) For any $0 < i < l$ it is possible to compute the fingerprint of $s_{1+i}, s_{2+i}, ...s_l$ $(\phi(s_{1+i}, s_{2+i}, ...s_l))$ from the fingerprint of $s_1, s_2, ...s_l$, $(\phi(s_1, s_2, ...s_l))$ and the fingerprint of $s_1, s_2, ...s_i$, $(\phi(s_1, s_2, ...s_i))$.

It is easy to see that the fingerprint from example 1 is not a Sliding Fingerprint.

Definition 3: Let $S = s_1, s_2, ...s_l$ and let $p \in \Theta(\mathbf{N}^4)$. Define the polynomial fingerprint of S to be $\phi(S)_{r,p} = (\sum_{i=1}^{l} s_i r^i) \bmod p$ for some random $r \in \mathbf{F_p}$.(Rabin and Karp [15], used this fingerprint)

Lemma 1: The polynomial fingerprint is a Sliding Fingerprint.

Proof: Lets $S = s_1, s_2, ...s_{l+1}$,$p \in \Theta(\mathbf{N}^4)$ and $r \in \mathbf{F_p}$.
The first condition is simple. Let q be the polynomial fingerprint of $s_1, s_2, ...s_l$.
By definition:

$$q = s_1 r + s_2 r^2 + ... + s_l r^l \bmod p$$

We add $s_{l+1} r^{l+1}$ to both sides of the equation to get:

$$q + s_{l+1} r^{l+1} = s_1 r + s_2 r^2 + ... + s_l r^l + s_l r^{l+1} \bmod p$$

This is exactly the fingerprint of $s_1, s_2, ...s_l, s_{l+1}$.

For the second condition, let q_1 be the polynomial fingerprint of $s_1, s_2, ...s_l$ and let q_2 be the polynomial fingerprint of $s_1, s_2, ...s_i$.
By definition:

$$q_1 = s_1 r + s_2 r^2 + ... + s_l r^l \bmod p \ \ and \ \ q_2 = s_1 r + s_2 r^2 + ... + s_i r^i \bmod p$$

By subtracting the two fingerprint and dividing them by r^i we get:

$$\tfrac{q_1 - q_2}{r^i} = s_{i+1} r^{i+1-i} + s_{i+2} r^{i+2-i} + ... + s_l^{l-i} = s_{i+1} r + s_{i+2} r^2 + ... + s_l^{l-i}$$

This is exactly the fingerprint of $s_{i+1}, s_{i+2}, ...s_l$. □

Theorem 1: Lets S_1 and S_2 be two strings of length $m \leq n$, $p \in \Theta(\mathbf{N}^4)$, and $\phi_{r,p}$ be a polynomial fingerprint. The probability that $\phi_{r,p}(S_1) = \phi_{r,p}(S_2)$ is less than $\frac{1}{n^3}$.

Proof: To satisfy the equation $\phi_{r,p}(S_1) = \phi_{r,p}(S_2)$, it is enough to satisfy $\phi_{r,p}(S_1) - \phi_{r,p}(S_2) = 0$. If we view this equation as a function on r, we get a polynomial modulo p with degree m. The number of $r \in \mathbf{F_p}$ that satisfy this equation is at most m. We choose r randomly from $\mathbf{F_p}$ so the probability we choose such an r is less than $\frac{m}{n^4} \leq \frac{n}{n^4} = \frac{1}{n^3}$. □

Denote by $|\phi|$, the length of the string that ϕ is calculated for.

Henceforth, when we speak of a Sliding Fingerprint, we refer to the polynomial fingerprint.

3. Related Work

Rabin and Karp [15] used fingerprinting in the late 80's to solve the pattern matching problem. Their main idea is very simple: at each text position they calculate the fingerprint of the last m characters and compere it to the fingerprint of the pattern. They use a Sliding Fingerprint in order to do this in linear time. As shown in section 2, in order to deterministically slide the fingerprint to the next position, they needed to save the last m character of the text.

Knuth, Morris and Pratt [13] were the first to achieve a deterministic linear time algorithm for this problem. The major insight that enabled them to get that result was using the information from the fact that position i matches the pattern in the first l character to facilitate the computation of the next position. In our work we will use one of their lemmas

In order to present the lemma we need to first define the shortest period.

Definition 4: Let $S = s_1, s_2, ...s_n$ be a string of length n. A prefix $S_p = s_1, s_2, ...s_l$ of S is define to be a period of S, iff $s_i = s_{i+l}$ for $0 \leq i \leq n - l$.

Lemma 2: Let $T = t_1, t_2, ...t_n$ be a text of length n, $P = p_1, p_2, ...p_m$ a pattern of length m, and let $1 \leq l \leq m$. We denote by $P_l = p_1, p_2, ...p_l$ the prefix of the pattern with length l. Denote the shortest period of P_l as $period_{P_l}$. If P_l matches the text at a given index i, then there can not be a match between position i to $i + |period_{P_l}|$.($|S|$ refer to the length of S)

Proof: By contradiction, assume that there is a match for a given index j such that $i < i + j < i + |period_{P_l}|$. P_l matches T at position i and at position $i+j$. From transitivity we get that P_l matches itself at position j. This means that $p_0, p_1, ...p_{j-1}$ is a period of the pattern, but $j < |period_{P_l}|$, in contradiction to the minimality of $period_{P_l}$. □

4. Our Method's Foundation

In this section we present an idea for solving the online pattern matching problem with minimal space. This idea is the heart of our algorithm. Our idea combines the key features of the KMP [13] algorithm and the Rabin-Karp [15] algorithm to achieve an online algorithm that uses less space.

As mentioned in the previous section, the problem with the Rabin-Karp algorithm is that it saves the last m characters in order to "slide" the fingerprint to the next position. It seems hard to slide the fingerprint without saving the last m characters so our method does not do it. When Rabin-Karp's algorithm is done with the i'th character, and advances to the next position in the text, it does not use any of the information gathered when processing the previous index. The KMP algorithm, on the other hand, puts that information to good use. For example, having a match at the i'th index indicates we know the last m characters, so there is no point in saving them. In addition, by the lemma 2 we know that there can't be another match until position $i + |period_P|$, where $period_p$ is the shortest period of P.

We present an idea for an algorithm that try to solve the pattern matching problem. This algorithm assumes a very stringent restriction whereby the following two conditions must be fulfilled in order to operate.

1) The pattern must match at the first position of the text.
2) The distance between every two successive match position is exactly $|period_P|$

In section 5 we extend our algorithm to handle all cases. (This algorithm actually verified that T is of the form $(period_P)^s$).

Algorithm - Verifying if $T = (period_P)^s$

- **In the preprocessing phase:** We calculate the Sliding Fingerprint on the pattern,ϕ_p and on the shortest period of the pattern,ϕ_{period_P}.
- **In the online phase:** we will slide a fingerprint, ϕ, over the entire text. For all places in which the fingerprint of the pattern is equal to ϕ we announce a 'match'; otherwise, we announce 'not match'. In order to slide the fingerprint over the text with less space we work as follows:
  - Calculate the fingerprint of the first m characters of the text, ϕ.
  - While $\phi = \phi_p$, slide ϕ by $|period_P|$ characters, by deducting ϕ_{period_P} from the fingerprint, and adding the next few symbols. (The pattern matches the text, and we know the fingerprint of the period of the pattern, hence we can slide.)
  - If we did not get to the end of the text, *abort*. (We do not handle this case.)

Theorem 2: If the algorithm does not return "abort", then the returned answer is correct:

1) For all i for which the algorithm announces 'match', there is indeed a match with high probability.
2) For all i for which the algorithm announces 'no match', with high probability there is no match.

Proof:

1) The proof follows from the definition of Sliding Fingerprint.
2) Lemma 2 tells us that if there is a match at a certain position, then there can not be a match at the next $|period_P|$ positions. This is the number of positions following a match where a fingerprint is not checked by our algorithm.
   The problem is, if we mistakenly announces a match, we will slide over $|period_p|$ position that could be a matches. However this case occur just after 'false positive', which with high probability does not happen. □

One problem with this suggested algorithm is that it requires the text and pattern to satisfy stringent restrictions that generally are not guaranteed.

5. Correct Algorithm

As mentioned above, our idea for solving the pattern matching problem with less space works only in the case where the distance between each two consecutive occurrences of the pattern in the text is exactly $|period_P|$, and the pattern matches the first text position. More formally, in order for our idea to work the following two conditions must be fulfilled:

1) The pattern must match at the first position of the text.
2) The distance between every two successive match positions is exactly $|period_P|$.

The reason for the first condition, is that our algorithm needs a starting point where the pattern matches the text. In order to solve this, we will consider $\log m$ subpatterns, $P_1, P_2, ...P_{\log m}$. i.e Let $P = p_1, p_2, ...p_m$ be the pattern, the i'th subpattern is defined as $P_i = p_1, p_2, ...p_{2^i}$. For each $0 \leq i \leq \log m$ we will denote the period of P_i by $period_{P_i}$. Our algorithm will try to find all the subpatterns instead of just the whole pattern. For its starting point it will be enough to find a position in which the smallest subpattern will match the text. It has length 1, hence it is very easy to find that position.

When our algorithm finds some position where P_i is a match, it tries to match P_{i+1} from the same starting point of P_i. If P_{i+1} is not match at that position, the algorithm will use the information that P_i is match in order to proceed. From the fact that P_i matches at that position, lemma 2 ensures us that P_i can not match at the next $|period_{P_i}|$ symbols, and hence neither can P_{i+1}. P_i can only match in a jump which is an integer factor of $|period_{P_i}|$ or after $|P_i|$ steps. P_{i+1} can not match in a location where P_i does not match. Therefore we want to check only in jumps of $|period_{P_i}|$ until there is no longer an overlap with the area where P_i matches. We know the fingerprint of the $period_{P_i}$, therefore we can slide by $|period_{P_i}|$ characters. In the case where P_{i+1} does not match, and we do not have an overlap with the P_i occurrence, we abort our algorithm.

This will be the "heart" of our algorithm, we refer to this algorithm as a "process". A process *lives* on a specific substring of the text, from the position it is started until it aborts.

In order to be able to find all the locations where the subpatterns match, we compute the Sliding Fingerprint on each subpattern, and the Sliding Fingerprint on its period, in the preprocessing stage. Consequently, overall we will need to save $2\log m$ Sliding Fingerprints.

Here is a formalization of how "process" work.

Process:

- Initialize an empty sliding fingerprint ϕ.
- For each character that arrive.
  - Extend ϕ to include the new character.
  - If $|\phi| = 2^i$ and $\phi \neq \phi_i$ for some $0 \leq i \leq \log m$.
    * If ϕ has at least $|period_{P_{i-1}}|$ length overlaps with the last match, slide ϕ by $|period_{P_{i-1}}|$ characters.
    * Else, *abort*

Lemma 3: A process finds all the matches between the pattern and the substring of his text with high probability.

Proof: Immediate from lemma 2 and theorem 2. □

In a location where our process aborts, we want to start a new process. The problem is what happens if there a match that starts in the substring of the first process and ends in the substring of the second process.

Definition 5: We define a text location to be a *checkpoint*, if there is $1 \leq i \leq \log m$ such that a matches of P_i to the text end at this location. We will also refer to this checkpoint by the length of the match, i.e. a 2^i-checkpoint.

Lemma 4: Let $T' = t_i, ...t_j$ be the substring of some process, and let τ be the index of the last checkpoint that the process found. There can not be a match that starts in location k, such that $i \leq k \leq \tau$, which is not contained in T'

Proof: Immediate from the way the process operates. □

From lemma 4, we know that we need to start a new process in the last checkpoint of each process. The problem is we don't know in advance that a specific checkpoint is the last of some process. So we will start a new process at every checkpoint. If our process will reach another checkpoint we will abort the process that was started on the previews checkpoint. This ensures that we start a new process on the last checkpoint of each process. Later we will prove that we will not have more then $O(\log m)$ "processes" in parallel.

When a process creates a new process, we refer to the new process as a “son process” of the creator process. We also refer to the creator process as a “father process” of the “son process”.

The outline of our algorithm is as follows:

Final - Algorithm

- **In the preprocessing:** Initialize an empty sliding fingerprint, ϕ, and for each $0 \leq i \leq \log m$, calculate the sliding fingerprint ϕ_i, of P_i, and the sliding fingerprint $\phi_{i,period}$, of the period of P_i.
- **In the online phase:**
  - Start a new process.
  - For any character that arrive, send it to all the process.
  - If some process 'abort' start new process.
  - If some process, A reach to a checkpoint:
    - Stop the 'son-process' of A (if has)
    - Start a new 'son-process' for A.

Theorem 3: The number of processes that run in parallel is no more then $3 \log m$

To prove the above theorem, we first prove a few lemmas.

Definition 6: We define $l-$process as a process that starts on some $l-$checkpoint.

We note that due to the way the algorithm works, each process has at most one “son process” at a given time.

Our proof contains two parts:

1) An $l-$process can not create a $2l-$process while its father is alive.
2) An $l-$process can not have a “great-grandson” that is also an $l-$process while it is alive.

Lemma 5: Define 'A' to be our main process, assume 'A' create an $l-$process 'B'. With at most $2l-1$ character from the checkpoint that created 'B'. One of the following two conditions must occur:

1) The process 'A' gets to a new checkpoint, and all of its descendants stop.
2) The process 'A' reaches 'abort'

Hence, process 'B' and 'A' can not live more then $2l-1$ characters in parallel.

Proof: After $2l-1$ characters, there are no overlaps between the last match and the fingerprint, ϕ, that A computes. This guarantees that after $2l-1$ characters, if 'A' does not get to a new checkpoint, it will abort. If it does get to a new checkpoint, it stops all of its descendant. □

Lemma 6: In order for a process to create an $l-$process, it must live at least l character.

Proof: Immediate from the definition of an $l-$process. An $l-$process is a process that is created on a checkpoint that was on a match of length l. In order for a process to get to a match of length l, it must see at least l characters. □

Lemma 7: An $l-$process can not create a $2l-$process while its father is alive.

Proof: We define 'A' and 'B' as before, We proved in lemma 5 that process 'B' and 'A' can not live more then $2l-1$ characters in parallel. We also proved in lemma 6 that in order for process 'B' to create a $2l$-process, it must live at least $2l$ characters. Hence, process 'B' can not create a $2l-$process while process 'A' is alive. □

Lemma 8: An $l-$process can not have a great-grandson that is also an $l-$process while it is alive.

Proof: Again, we define 'A' and 'B' as before. Assume 'A' is the first process of length l. We prove that 'A' and 'B' live at most $2l-1$ characters in parallel. So, 'B' can also create an $l-$process, 'C', as a ”son process”. However, in order for 'C' to also create an $l-$ process, it must wait l characters from the time it was created. overall $2l$ characters from the time 'B' was created. This means 'A' is no longer alive. □

Conclusion 1: The number of processes that run in parallel is no more then $3 \log m$

Proof: Hence we have just $\log m$ available length for each process, and we prove that there are at most 3 processes for each length. We get that there are less then $3 \log m$ processes working in parallel. □

Theorem 4:

- The space complexity of our algorithm is $O(\log m)$.
- The running time of our algorithm is bounded by $O(\log m)$ per character.

Proof:

- **Space Complexity:** All the fingerprints from our preprocessing use $O(\log m)$ space. In addition, each process saves another fingerprint. We have at most $\log m$ processes in parallel, hence we save in addition at most $\log m$ fingerprints. Overall we use $O(\log m)$ space.
- **Running Time:** Each process spends $O(1)$ time for each new character that arrives, and each time there are at most $3 \log m$ processes running. Thus, overall, the running time of our algorithm is $O(\log m)$ per character. □

Theorem 5: The probability that the algorithm gives a false answer is less then $\frac{1}{n^2}$

Proof: Our algorithm announces a match only after the fingerprint of the pattern matches the fingerprint of the last m text characters. We have already shown that the probability of two different string having the same fingerprint is less then $\frac{1}{n^3}$. We calculate the fingerprint on n strings, for each position. Using the union bound, the probability to get a wrong answer in at least one index is less then the sum of the probabilities of an error in each position. Overall the error probability is less then $\frac{1}{n^2}$. □

6. Pattern Matching with 1-Mismatch

In this section we present an online algorithm for finding all the text location that match the pattern with exactly one mistake. Our solution uses the online exact pattern matching algorithm as a black-box. We denote the online exact pattern matching algorithm by Exact_PM.

Lets $T = t_0, t_1, ...t_{n-1}$ be the text and $P = p_0, p_1, ...p_{m-1}$ be the pattern, and let $q_1, q_2, ...q_l$ be l prime numbers such that $\prod_{i=1}^{l} q_i > m$. We will create l groups of partitions to the text in the following way. For the i'th group, we will partition the text to q_i partitions. The j'th partition of the i'th group of the text will be denoted by $T_{q_i,j}$. It will consist all characters t_τ such that $\tau \bmod q_i = j$. For example, for $q_i = 2$ the group will consist both $T_{2,0} = t_0, t_2, t_4...$ and $T_{2,1} = t_1, t_3, t_5...$. We partition the pattern in the same way.

At every text location we want to match every partition of the pattern to its corresponding text partition. The problem is that in the τ'th position of the text, the corresponding partition of some subpattern is not the same as in the $\tau + 1$'th position. i.e the corresponding partition of $P_{2,0} = p_0, p_2, ...p_{2i}...$ at the m'th position of the text (m is even) is $t_0, t_2, ...t_{2i}...$ but at the $m + 1$ position of the text is $t_1, t_3, ...t_{2i+1}...$. If we just align $P_{2,0}$ with $T_{2,0}$ we will not got the the different between $p_0, p_2, ...p_m$ and $t_1, t_3, ...t_{m+1}$ that is needed at the $m + 1$ position. So we need to align every partition of the pattern, $P_{q_i,j}$, to q_i text shifts.

For each pattern partition $P_{q_i,j}$, run q_i processes of Exact_PM. Denote the σ'th process of the subpattern $P_{q_i,j}$, for $0 \le \sigma < q_i$, by $Process_{q_i,j,\sigma}$. $Process_{q_i,j,\sigma}$ will try to match $P_{q_i,j}$ to the text by considering the text as if it starts from the σ character. More precisely, when the τ'th character of the text arrives, it goes as input to the process $Process_{q_i,j,\sigma}$ only if $\tau \bmod q_i = j - \sigma$.

For a specific alignment of the pattern and the text, only one of the shifts of each specific partition of the pattern $P_{q_i,j}$ interests us. i.e in the m position we interest to match $T_{2,0}$ with $P_{2,0}$ and $T_{2,1}$ with $P_{2,1}$ but not $T_{2,0}$ with $P_{2,1}$. For each q_i and σ, denote by $numOfNotMatch_{q_i,\sigma}$ the number of j's for which $P_{q_i,j}$ does not match in this alignment. In alignment that end in the τ character, $\sigma = ((\tau + 1) \bmod m) \bmod q_i$ for all $q_i \in \{q_1, ...q_l\}$.

For each alignment, if for all q_i, $numOfNotMatch_{q_i,\sigma} = 0$, we will announce a 'match'. Otherwise if for all q_i, $numOfNotMatch_{q_i,\sigma} = 1$, we will announce 'there is exactly 1-mismatch'. Otherwise, we will announce 'more than 1-mismatch'.

More formally the outline of our algorithm is as follow:

Algorithm - 1-mismatch

- **In the preprocessing:** Initialize a process of Exact_PM, for each $q_i \in \{q_1, q_2, ...q_l\}$, $0 \le j < q_i$ and $0 \le \sigma < q_i$. We denote each such process by $Process_{q_i,j,\sigma}$.
- **In the online phase:** Send the τ character to each $Process_{q_i,j,\sigma}$ such that $\tau \bmod q_i = j - \sigma$.
  - If some $Process_{q_i,j,\sigma}$ return 'not match'
    - $numOfNotMatch_{q_i,\sigma} = numOfNotMatch_{q_i,\sigma} + 1$
  - At the end of each position τ, let $\sigma = ((\tau + 1) \bmod m) \bmod q_i$.
    - If $numOfNotMatch_{q_i,\sigma} = 0$ for all $q_i \in \{q_1, ...q_l\}$. - announce a 'match'
    - If $numOfNotMatch_{q_i,\sigma} = 1$ for all $q_i \in \{q_1, ...q_l\}$. - announce 'there is exactly 1-mismatch'
    - Else - announce 'more than 1-mismatch'.
    - assign 0 to $numOfNotMatch_{q_i,\sigma}$ for all $q_i \in \{q_1, ...q_l\}$. (for the next alignment)

We had q_i partitions that needed q_i shifts for each i. So overall we run $\sum_{i=1}^{l} q_i^2$ processes of Exact_PM.

At each text location τ, for each $q_i \in \{q_1, q_2, ...q_l\}$ and $0 \le j < q_i$ there is exactly one σ such that $\tau \bmod q_i = j - \sigma$. Therefore no symbol can go as input to two difference processes with the same subpattern. There are exactly $\sum_{i=1}^{l} q_i$ different subpatterns. So overall we run Exact_PM $\sum_{i=1}^{l} q_i$ times at each character.

Lemma 9: There exists a constant c such that for any x, there exist $\frac{x}{\log m}$ prime numbers, between x, and cx.

Lemma 10: Let $q_1, q_2, q_3...q_{\frac{\log m}{\log\log m}}$ be $\frac{\log m}{\log\log m}$ prime numbers between $\log m$ and $c \log m$ for some constant c. Then $\prod_{i=1}^{\frac{\log m}{\log\log m}} q_i > m$.

Proof: For all i, $q_i \ge \log m$. Hence $\prod_{i=1}^{\frac{\log m}{\log\log m}} q_i > \prod_{i=1}^{\frac{\log m}{\log\log m}} \log m = \log m^{\frac{\log m}{\log\log m}} = m$. We got, $\prod_{i=1}^{\frac{\log m}{\log\log m}} q_i > m$ □

Lemma 11: Let $q_1, q_2, q_3...q_{\frac{\log m}{\log\log m}}$ be $\frac{\log m}{\log\log m}$ prime numbers between $\log m$ and $c \log m$ for some constant c. Then, $\sum_{i=1}^{\frac{\log m}{\log\log m}} q_i \in O(\frac{\log^2 m}{\log\log m})$ and $\sum_{i=1}^{\frac{\log m}{\log\log m}} q_i^2 \in O(\frac{\log^3 m}{\log\log m})$.

Proof: For every i, $q_i \le c \log m$. Hence $\sum_{i=1}^{\frac{\log m}{\log\log m}} q_i \le \sum_{i=1}^{\frac{\log m}{\log\log m}} c \log m = \frac{c \log^2 m}{\log\log m} = O(\frac{\log^2 m}{\log\log m})$.

For every i, $q_i \le c \log m$, and $q_i^2 \le c^2 \log^2 m$. Hence $\sum_{i=1}^{\frac{\log m}{\log\log m}} q_i^2 \le \sum_{i=1}^{\frac{\log m}{\log\log m}} c^2 \log^2 m = \frac{c^2 \log^3 m}{\log\log m} = O(\frac{\log^3 m}{\log\log m})$.

□

Lemma 12: At a specific alignment of the pattern with the text, the number of mismatches is exactly one iff for every q_i all the subpatterns $P_{q_i,j}$, $0 \le j < q_i$ match at this alignment, except for exactly one.

Proof: First, If two subpatterns $P_{q_i,j}, P_{q_i,j'}$ are not matches then it immediately yields more then 1-mismatch. In addition, if for all j, $P_{q_i,j}$ matches at this alignment, then there is a match at this position.

Assume, in contradiction, that there exists more then one mismatch but for every q_i there is no more than one subpattern $P_{q_i,j}$ that does not match. Consider two mismatches at positions i_1, i_2, for all q_i they both map to the same subpattern $P_{q_i,j}$. otherwise there is more then one subpattern $P_{q_i,j}$ that does not match. Consequently $i_1 \bmod q_i = i_2 \bmod q_i$ for every q_i. Hence $\prod_{i=1}^{l} q_i > m$ By the Chinese Remainder Theorem we get $i_1 = i_2$, a contradiction. □

Lemma 13: If there is exactly 1 mismatch then our algorithm easily finds the exact position that mismatch.

Proof: Immediate from the Chinese Remainder Theorem. □

Theorem 6:

- The space complexity of the $1-$mismatch algorithm is $O(\frac{\log^4 m}{\log\log m})$.
- The running time of the $1-$mismatch algorithm is bounded by $O(\frac{\log^3 m}{\log\log m})$ per character.

Proof:

- **Space Complexity:** we run in parallel $\sum_{i=1}^{\log m} q_i^2 \in O(\frac{\log^3 m}{\log\log m})$ processes of Exact_PM. each process take $O(\log m)$ space. Therefore the total amount of space used is $O(\frac{\log^4 m}{\log\log m})$.
- **Running Time:** As was shown earlier, the number of processes of Exact_PM that run on each character is bounded by $\sum_{i=1}^{\log m} q_i \in O(\frac{\log^2 m}{\log\log m})$. The running time for each character of the Exact_PM is bounded by $O(\log m)$. Overall we get that the running time for each character is bounded by $O(\frac{\log^3 m}{\log\log m})$. □

7. PATTERN MATCHING WITH k-ERRORS

In this section we solve the pattern matching with k errors problem. The time complexity of our solution is bounded by $O(k^2 poly(\log m))$ per character. Our algorithm uses group testing combined with the 1-mismatch algorithm from section 6.

The group testing problem can be described as follows. Consider a set of n items, each of which can be defective or non-defective. The task is to identify the defective items using a minimum number of tests. Each test works on a group of items simultaneously and returns whether that group contains at least one defective item or not. If a defective item is present in a group then the test result is said to be positive, otherwise it is negative.

Group testing has a long history dating back to at least 1943 [9]. In this early work the problem of detecting syphilitic men for induction into the United State military using the minimum number of laboratory tests was considered. Subsequently a large literature has built up around the subject and we refer the interested reader to [8] for a comprehensive survey of the topic. We use a slightly different version of the group testing problem. In our version the tests return positive, if there is exactly one defective item in the group, and negative otherwise. We obtain the test result by the 1-mismatch algorithm presented in section 6.

A group is actually a partition of the string. We say that p_i is in some group, if a part in that group contains p_i.

In the remainder of this section we show how the groups are chosen, and how the number of mismatches up to k at every location can be determined as a result of our group testing.

Let $q_1, q_2, ...q_l$ be l prime numbers such that $\prod_{i=1}^{l} q_i > m^k$. **Exactly as in the previous section**, construct l partition groups of the text in the following way. For the i'th group, we will partition the text into q_i partitions. The j partition of the i'th group contains all the characters τ such that $\tau \bmod q_i = j$. Overall we build $\sum_{i=1}^{l} q_i$ partitions of the text. We partition the pattern in the same way.

As seen in section 6, at every position there is exactly one partition of the pattern that corresponds to each partition of the text. Each such couple will be a group, in the group testing problem. From the previous section, we know how we can match each couple. Now we will use the 1-mismatch algorithm as a black-box instead of the exact pattern matching algorithm. The time for each character will be bounded by $(\sum_{i=1}^{l} q_i) \cdot O(\frac{\log^3 m}{\log\log m})$.

As in section 6 we need to match each partition of the pattern $P_{q_i,l}$ in q_i shifts. So overall the space complexity will be $(\sum_{i=1}^{l} q_i^2) \cdot O(\frac{\log^4 m}{\log\log m})$.

The outline of our algorithm is as follow:

Algorithm - $k-$errors

- **In the preprocessing:** Initialize a process of $1-$mismatch, for each $q_i \in \{q_1, q_2, ...q_l\}$, $0 \leq j < q_i$ and $0 \leq \sigma < q_i$. we denote each such process by $Process_{q_i,j,\sigma}$.
- **In the online phase:** Send the τ character to each $Process_{q_i,j,\sigma}$ such that $\tau \bmod q_i = j - \sigma$.
- At the end of each alignment:
  - Exclude all the mismatch from all the process that return 'there is exactly 1-mismatch'. Denote their number by d
  - If $d > k$ - announce 'their is more than k mismatches'.
  - Else if all other mismatches can explain by the d mismatches we found, announce 'd-mismatches'.
  - Else announce 'their is more than k mismatches'

Lemma 14: Let p_j be some character from the pattern. At any alignment between the pattern and the text. For each group of k indices, $i_1, i_2, ..., i_k$, and for each $1 \leq j < m$. p_j maps at this location to at least one partition where $i_1, i_2, ..., i_k$ does not occur.

Proof: In contradiction, let assume that there exist some group $i_1, i_2, ..., i_k$ and a position j, such that **for every partition** that j belongs to there exist some $\tau \in i_1, i_2, ..., i_k$ that belong to the same partition. Each partition is associate with some prime number $q_i \in \{q_1, q_2, ...q_l\}$, and $\prod_{i=1}^{l} q_i > m^k$. From the pigeonhole principle, there is some $\tau \in i_1, i_2, ..., i_k$ such that, τ and j belong together to e partition such that, $\prod_{i=1}^{e} q_i \geq m$. From the Chinese Reminder Theorem we have that $j = \tau$, in contradiction. □

Conclusion 2: If there are at most k mismatches, each mismatch will be mapped to at least one partition where all the other mismatches do not belong. So, using the 1-mismatch algorithm on these partitions, will discover all the mismatches.

Lemma 15: It is possible to observe if the pattern does not match up to k error at each alignment.

Proof: For a specific alignment. Let d be the number of partitions that had exactly one mismatch, and let $d' > k$ be the number of mismatches.

- If $d > k$, it means that we have found more then k mismatches. Hence we observe that the pattern does not match at this location with up to k errors.
- If $d \leq k$, let $i_1, i_2, ...i_d$ be the d locations where we found a mismatch, and let j be some position of a mismatch that we have not found. From Lemma 14 we know that there will be a partition that contains index j but dos not contain any of the the d mismatches that we noticed. In this partition there are more then one mismatch, otherwise we would notice it. Hence we will not be able to explain why there is a mismatch in that partition. But from conclusion 2 we know that if there are less then k mismatches, we will be able to explain every partition that does not match. Hence we observed that the pattern does not match up to k errors at this location.

□

Lemma 16: Let $q_1, q_2, ...q_l$ be l prime numbers greater then $\log m$. In order for $\prod_{i=1}^{l} q_i > m^k$, it is necessary that $l \in O(\frac{k \log m}{\log \log m})$

Proof: For all i, $q_i \geq \log m$, hence $\prod_{i=1}^{l} q_i \geq \prod_{i=1}^{l} \log m = \log^l m$. Using $l = \frac{k \log m}{\log \log m}$ we have $\log m^{\frac{k \log m}{\log \log m}} = m^k$. □

Lemma 17: Let $q_1, q_2, q_3 ... q_{\frac{k \log m}{\log \log m}}$ be $\frac{k \log m}{\log \log m}$ prime number between $\log m$ and $ck \log m$ for some constant c. Then $\sum_{i=1}^{\frac{k \log m}{\log \log m}} q_i \in O(\frac{k^2 \log^2 m}{\log \log m})$ and $\sum_{i=1}^{\frac{k \log m}{\log \log m}} q_i^2 \in O(k^3 \frac{\log^3 m}{\log \log m})$.

Proof: For every i, $q_i \leq ck \log m$. Hence $\sum_{i=1}^{\frac{k \log m}{\log \log m}} q_i \leq \sum_{i=1}^{\frac{k \log m}{\log \log m}} ck \log m = \frac{ck^2 \log^2 m}{\log \log m} = O(k^2 \frac{\log^2 m}{\log \log m})$

For every i, $q_i \leq ck \log m$. Hence $\sum_{i=1}^{\frac{k \log m}{\log \log m}} q_i^2 \leq \sum_{i=1}^{\frac{k \log m}{\log \log m}} (ck \log m)^2 = \frac{c^2 k^3 \log^3 m}{\log \log m} = O(k^3 \frac{\log^3 m}{\log \log m})$

□

Theorem 7:

- The space complexity of the $k-$mismatch algorithm is $O(k^3 poly(\log m))$.
- The running time of the $k-$mismatch algorithm is bounded by $O(k^2 poly(\log m))$ per character.

Proof:

- **Space Complexity:** We run in parallel $\sum_{i=1}^{k \log m} q_i^2 \in O(k^3 \frac{\log^3 m}{\log \log m})$ processes of the 1-mismatch algorithm. Each process requires $\log^4 m$ space, making the overall space requirement $O(k^3 poly(\log m))$.
- **Running Time:** As we have shown, the number of processes of the 1-mismatch algorithm for each character is bounded by $\sum_{i=1}^{k \log m} q_i \in O(k^2 \frac{\log^2 m}{\log \log m})$. The running time for each character of the 1-mismatch algorithm is bounded by $O(\log^3 m)$. Overall the time required per character is bounded by $O(k^2 poly(\log m))$. □

8. Conclusion and open problem

We suggested an online algorithm that solves the pattern matching problem with $O(\log m)$ space. The running time of our algorithm is bounded by $O(\log m)$ per character. Also we present an online algorithm for the pattern matching with k-mismatches problem that uses $O(k^3 poly(\log m))$ space, and the running time per character is bounded by $O(k^2 poly(\log m))$.

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

[2] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In *J. of Algorithms*, pages 257–275, 2004.

[3] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[4] R. Clifford, K. Efremenko, B. Porat, and E. Porat. A black box for online approximate pattern matching. *Proc. of the Symposium on Combinatorial Pattern Matching (CPM)*, pages 143–151, 2008.

[5] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, 1991.

[6] M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string-searching. *Algorithmica*, 13(5):405–425, 1995.

[7] M. Crochemore. String-matching on ordered alphabets. *Theor. Comput. Sci.*, 92(1):33–47, 1992.

[8] D.Z. Du and F.K. Hwang. *Combinatorial Group Testing and its Applications*, volume 12 of *Series on Applied Mathematics*. World Scientific, 2nd edition, 2000.

[9] R. Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

[10] L. Gasieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theor. Comput. Sci.*, 147(1-2):19–30, 1995.

[11] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 106–113, New York, NY, USA, 1981. ACM.

[12] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. pages 166–173, 1998.

[13] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[14] S. Rao Kosaraju. Efficient string matching. Manuscript, 1987.

[15] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Res. and Dev.*, pages 249–260, 1987.

[16] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[17] W. Rytter. On maximal suffixes and constant-space linear-time versions of kmp algorithm. *Theor. Comput. Sci.*, 299(1-3):763–774, 2003.