Theoretical Computer Science 638 (2016) 3–10

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Set Intersection and Sequence Matching with mismatch counting ☆

Ariel Shiftan *, Ely Porat *

Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel

ARTICLE INFO

Article history:
Received 2 April 2015
Received in revised form 3 January 2016
Accepted 7 January 2016
Available online 12 January 2016

Keywords:
Set Intersection Matching
Sequence Matching
Pattern matching
Generalized strings

ABSTRACT

In the classical pattern-matching problem, one is given a text and a pattern both of which are sequences of letters. The requirement is to find all occurrences of the pattern in the text. We studied two modifications of the classical problem, where each letter in the text and pattern is a set (*Set Intersection Matching* problem) or a sequence (*Sequence Matching* problem). Two "letters" are found to match if the intersection of the corresponding sets is not empty or if the two sequences have a common element in the same index. We first show that the two problems are similar by establishing a linear time reduction between them. We then show the first known non-trivial and efficient algorithms for these problems, when the maximum set/sequence size d is small. The first is a Monte Carlo randomized algorithm for *Set Intersection Matching*, that takes $\Theta\left(4^d n \log n \log m\right)$ time, where n and m are the lengths of the text and the pattern, respectively; the failure probability is less than $\frac{1}{n^2}$. This algorithm can also be used, with slight modifications, when up to k mismatches is allowed. In addition, it can be used to maintain an approximation of factor $1 \pm \epsilon$ of the mismatch count in $\Theta\left(\frac{1}{\epsilon^2} 4^d n \log n \log m\right)$ time; the failure probability is bounded by $\frac{1}{n}$. The second is a deterministic algorithm for *Set Intersection Matching* that can be used to count the number of matches at each index of the text in a total running time $\Theta\left(\sum_{i=1}^{d} \binom{\sigma}{i} n \log m\right) = O\left(\sigma^d n \log m\right)$, where σ is the size of the alphabet. The third algorithm, also deterministic, solves the *Sequence Matching* problem in $\Theta\left(4^d n \log m\right)$ time.

1. Introduction and related work

The classic pattern matching problem is defined as follows: We are given a text $T = t_0, t_1, t_2, \ldots, t_{n-1}$ of size n and a pattern $P = p_0, p_1, p_2, \ldots, p_{m-1}$ of size m — both are sequences of letters belonging to a pre-defined set, i.e., the alphabet Σ of size σ. We are required to find all occurrences of the pattern in the text. Linear time solutions were given in [10,7].

Two forms of approximation for the problem that are commonly researched involve *don't cares* and *mismatch count*. The first is the presence of a wildcard letter that matches any other letter. One of the main approaches for solving that problem, that was first introduced by Fischer and Paterson, is to use convolutions. They showed how the problem can be

☆ This work is part of Ariel Shiftan's PhD thesis done in the faculty of Computer Science, Bar-Ilan University.

* Corresponding authors.
E-mail addresses: shiftaa@cs.biu.ac.il (A. Shiftan), porately@cs.biu.ac.il (E. Porat).

http://dx.doi.org/10.1016/j.tcs.2016.01.003
0304-3975/

solved in $\Theta(n \log m \log \sigma)$ time [4]. Since then many algorithms [11–14] showed how different string matching problems can be solved efficiently by using FFT. Faster algorithms for the string matching with wildcards were presented later, and the time was reduced to $\Theta(n \log m)$ [9,8]. The second form of approximation is match/mismatch count, e.g. the number of matching letters when comparing the pattern to the text at location i for all locations. This mismatch count, at each location compared, is also known as the Hamming distance between the text and the pattern at this location. A well known algorithm for finding the hamming in $\Theta(n\sigma \log m)$ time was given in [4].

The subset matching problem was first introduced by Cole and Hariharan [1]. It extends the classic string-matching problem and defines both the pattern and text to be sequences of sets of characters. Formally, each text location t_i and each pattern location p_i is a set of characters, not a single character, taken from a certain alphabet. Pattern P is said to match text T at location i, if $p_j \subseteq t_{i+j}$ for all j, $0 \leq j < m$. Cole and Hariharan proposed a near-linear time randomized algorithm [1] and improved it [2] to a deterministic one. Amihood, Porat and Lewenstein proposed an approximate version with don't cares [3].

Generalized strings are another extensions of the classical string-matching problem. Assuming the alphabet is Σ, *Generalized strings* are sequences of sets, where each set is a subset of Σ that are possibilities for letters in that position. For example, the *generalized string* [{a,b},{b,d}] matches the strings ab, ad, bb, and bd. This type of string is used in [6] and while specifying the pattern to look for regular expressions.

As in the subset matching and *Generalized strings* problems, the two problems studied in this paper define the pattern and the text to be sequences of sets (or sequences). But here, a match between two 'letters' is when either the intersection of the two sets is not empty, as in the *Set Intersection Matching* problem, or when the two sequences have a common element in the same place, as in the *Sequence Matching* problem. In the former, a search is conducted to find one *generalized string* inside another. Assuming the maximum set (or sequence) size is d, it is trivial that these problems can be easily solved in $\Theta\left(nmd^2\right)$ and $\Theta(nmd)$ time, respectively.

Sample applications for the algorithms in this paper occur when there are possible errors in both the pattern and the text. For example, consider the case when both the pattern and the text were acquired using an Optical Character Recognition (OCR) algorithm that reports few options for each letter. One can create a set of all possible letters for each such letter and then run a *Set Intersection Matching* algorithm to get the occurrences of such faulty patterns in the text.

1.1. Our contribution

The difficulty in both problems lies in the lack of the transitive property, which is the base for fast pattern-matching algorithms. The transitive relation states that if $a = b$ and $b = c$, then $a = c$. This is utilized by classical pattern-matching algorithms to avoid comparing elements in which matching could be concluded. Because this property does not hold in the *Set Intersection Matching* and *Sequence Matching* problems, the existing algorithms cannot be used.

In this paper, we show the first known non-trivial and efficient algorithms for these problems when the maximum set/sequence size d is small. First, we show a Monte Carlo algorithm for *Set Intersection Matching* that takes $\Theta\left(4^d n \log n \log m\right)$ time. It has a failure probability less than $\frac{1}{n^2}$. This algorithm can be used to maintain an approximation of factor $1 \pm \epsilon$ of the mismatch count in $\Theta\left(\frac{1}{\epsilon^2} 4^d n \log n \log m\right)$ time. The failure probability is bounded by $\frac{1}{n}$. It improves upon the trivial solution where $d - 2\log d < \log m - \log\log n - \log\log m$. Assuming $\log n = o(m)$, it improves for $d < \frac{1}{2}\log m$. Next, we present a deterministic algorithm for *Set Intersection Matching*, that can be used to count the number of matches at each index of the text in a total running time $\Theta\left(\sum_{i=1}^{d} \binom{\sigma}{i} n \log m\right) = O\left(\sigma^d n \log m\right)$. This algorithm improves upon the trivial solution where $d < \frac{\log m}{\log \sigma}$. Finally, a $\Theta\left(4^d n \log m\right)$ time deterministic algorithm for the *Sequence Matching* problem is given, which improves upon the trivial algorithm for $d < \frac{1}{2}\log m$.

1.2. Overview and comparison

The structure of this paper is as follows. Section 2 formally defines each of the two problems. Sections 3 and 4 present three algorithms for the problems.

The two algorithms presented in Section 3 are for the *Set Intersection Matching* problem, and both can be used for determining the mismatch count. The first is randomized, approximates the mismatch count with high probability, and is more suitable for cases where the size of the alphabet σ is large, whereas the second is deterministic and gives the exact mismatch count but is less efficient when σ is large. The algorithm given in Section 4 is for the *Sequence Matching* problem, it is deterministic, and efficient for a large alphabet σ, but cannot be used to determine the mismatch count.

2. Preliminary

In the *Set Intersection Matching* problem, each pattern and text location is a set, and two locations match if the intersection between the sets is not empty. More formally:

Table 1
Set intersection example.

Index (i)	Intersection (j)			Match
	0	1	2	
0	{a,b}	{c}	{d}	√
1	{b}	{a}	{c,d}	√
2	{a}	{a,c}	∅	×

Table 2
Sequence matching example.

Index (i)	Matching (j)			Match
	0	1	2	
0	[1,1]	[0,1,0]	[0,1]	√
1	[0,0,0]	[1,0]	[0,0,0]	×
2	[1,0]	[1,1,0]	[0]	×

Definition 1 *(The Set Intersection Matching problem).*
Input:

- Alphabet Σ of size σ.
- Text $T = t_0, t_1, t_2, \ldots, t_{n-1}$, where each t_i is a set of letters from the alphabet Σ, and $|t_i|$ is the size of the set t_i.
- Pattern $P = p_0, p_1, p_2, \ldots, p_{m-1}$, where each p_i is a set of letters from the alphabet Σ, and $|p_i|$ is the size of the set p_i.
- d — maximum set size. Formally: $d = \max\left(\max_{i\in[1,m]}|p_i|, \max_{j\in[1,n]}|t_j|\right)$.

Output: All locations $0 \le i \le n-m$ such that for all $0 \le j \le m-1$: $t_{i+j} \cap p_j \neq \emptyset$.

Example 1. Consider the text T = [{a,b}, {b,c,d}, {a,d}, {a,c,d}, {b}] and the pattern P = [{a,b}, {a,c}, {c,d}]. The matches for all possible locations in the text are described in Table 1.

Note that the above definition ignores the order of the elements inside each set. In contrast, the *Sequence Matching* problem does consider the order. In this problem, both the pattern and the text are composed of sequences of letters from the alphabet, and a match between the two sequences occur when there is at least one index where both sequences have the same letter. Formally:

Definition 2 *(The Sequence Matching problem).*
Input:

- Alphabet Σ of size σ.
- Text $T = t_0, t_1, t_2, \ldots, t_{n-1}$, where each t_i is a sequence of letters from the alphabet Σ, and $|t_i|$ is the size of the sequence t_i.
- Pattern $P = p_0, p_1, p_2, \ldots, p_{m-1}$, where each p_i is a sequence of letters from the alphabet Σ, and $|p_i|$ is the size of the sequence p_i.
- d — maximum sequence size. Formally: $d = \max\left(\max_{i\in[1,m]}|p_i|, \max_{j\in[1,n]}|t_j|\right)$.

Output: All locations $0 \le i \le n-m$, such that for all $0 \le j \le m-1$, there exists $1 \le k \le \min\left(|t_{i+j}|, |p_j|\right)$: $t_{i+j}[k] = p_j[k]$.

Example 2. Consider the text T = [[a,b], [b,c,d], [a,d], [a,c,d], [b]] and the pattern P = [[a,b], [a,c], [c,d]]. The matches for all possible locations in the text are described in Table 2.

2.1. Linear time reduction

We now show a reduction between the two problem which demonstrates that the two may be equivalent except for a small difference in efficiency. Thus, giving an efficient solution to one may be sufficient to solve both efficiently.

To reduce *Sequence Matching* to *Set Intersection Matching* one can replace each of the elements in each sequence with a couple representing the element and its index. For example: $[[a,b],[b,c,d]] \Rightarrow [\{(a,1),(b,2)\},\{(b,1),(c,2),(d,2)\}]$. By doing so, the intersection can contain only elements that are in the same index, which is actually the definition of the *Sequence Matching* problem. The price we pay for this is the enlargement of the alphabet size to at most $d\sigma$.

To reduce *Set Intersection Matching* to *Sequence Matching* one should first repeat the set of letters of each text location d times. Then, pad the pattern set of letters to be of size d and repeat the new set (including the padding) d times. This

causes the intersection to appear in the same index of both the pattern and the text. For example: $T = [\{a, b\}, \{b, c, d\}]$, $P = [\{a, b\}, \{a, c\}] \Rightarrow T = [[a, a, a, b, b, b], [b, b, b, c, c, c, d, d, d]]$, $P = [[a, b, X, a, b, X, a, b, X], [a, c, X, a, c, X, a, c, X]]$ for $d = 3$ and X a padding letter. In this case the price is the squaring of d.

3. Set Intersection Matching algorithms

In this section, we show two algorithms for the *Set Intersection Matching* problem: The first is a Monte Carlo randomized algorithm that can also approximate the mismatch count. The second is a deterministic algorithm that is also able to count mismatches.

3.1. Randomized algorithm

We now show a Monte Carlo randomized algorithm for the problem. First, we give a 'bad' randomized algorithm — one that has a high probability of being wrong. We reduce this failure probability by running the algorithm several times.

The algorithm works as follows: First, we choose a random hash function $h : \Sigma \to \{0, 1\}$, a pair-wise independent function is sufficient. Then, we build a new text T_h using a linear phase that scans the original text T, and writes at position i of T_h one of 0, 1 or ϕ (where ϕ stands for don't care) depending on the set of letters at t_i. We write 0 if $h(x) = 0$ for all $x \in t_i$, 1 if $h(x) = 1$ for all $x \in t_i$, and ϕ if there exists $x, y \in t_i$ such that $h(x) = 0$ and $h(y) = 1$. We build P_h from the original pattern P in the same way.

The idea of the algorithm is based on the following two trivial lemmata:

Lemma 3. *If $t_{i+j} \cap p_j \neq \emptyset$ then for any hash function, the resulting letters will not mismatch.*

Lemma 4. *If $t_{i+j} \cap p_j = \emptyset$ the probability that these two places will not match is $2^{-(|t_{i+j}|+|p_j|)+1}$.*

Our 'bad' random algorithm will run regular pattern-matching algorithms on a binary alphabet to match between T_h and P_h. If we only want to test for match or not, we could use two convolutions (which are $O(n \log m)$). If we are also interested in up to k errors (up to k sets that do not intersect), we can run the algorithm from [5] that will return the places of the mismatch in time $O(nk \log^2 m)$. Note that the algorithm, in either case, fails to return the true answer with high probability.

Lemma 5. *If we run the 'bad' random algorithm $\frac{3}{2}4^d \ln n$ times the failure probability of the algorithm will be less than $\frac{1}{n}$.*

Proof. We deal with the following cases separately: Testing for an intersection match or not, and counting the number of places that do not intersect up to k.

In the first case, we examine a specific shift in which we check whether the pattern matches the text. If the pattern matches at this position, our algorithm will always return that it matches. If the pattern does not match on that location, it does not match with a probability higher than 2^{-2d+1} in each iteration. Therefore, with probability values less than $(1 - 2^{-2d+1})^{\frac{3}{2}4^d \ln n} \leq \frac{1}{n^3}$ we will obtain a false match. As we do the same for less than n shifts, the overall probability that there exists a shift in which our algorithm fails is bounded by $\frac{1}{n^2}$.

If we want to count the non-intersecting groups up to k, assume that $t_{i+j} \cap p_j = \emptyset$. The probability to miss it at each iteration is less than $1 - 2^{-2d+1}$. Therefore the probability that we will miss it in all of the iterations is bounded by $(1 - 2^{-2d+1})^{\frac{3}{2}4^d \ln n} \leq \frac{1}{n^3}$. We have at most nm pairs for which $t_{i+j} \cap p_j = \emptyset$. Hence, the probability to miss one of them is bounded by $\frac{m}{n^2} < \frac{1}{n}$. □

Lemma 6. *If we run the 'bad' random algorithm $\frac{4}{\epsilon^2}2^{2d-1} \ln n$ times, and calculate the total mismatch count and divide it by $\frac{4}{\epsilon^2} \ln n$, we get the $1 \pm \epsilon$ approximation factor to the mismatch count with probability higher than $\frac{1}{n}$.*

Proof. We look at a specific shift in which we check the mismatch count between the pattern and the text, and denote the mismatch count with β. β is an integer summation of the mismatches of pattern and text letters at this shift. If a pattern letter matches a text letter, a mismatch will never be counted. Otherwise, it will be reported as a mismatch with probability higher than 2^{-2d+1}. We define X_i as a random variable of the ith round mismatch count, and $X = \Sigma_{i=0}^{\frac{4}{\epsilon^2}2^{2d-1} \ln n} X_i \frac{1}{\beta}$. Note that $0 \leq \frac{X_i}{\beta} \leq 1$, $E(\frac{X_i}{\beta}) = 2^{-2d+1}$ and $E(X) = \frac{4}{\epsilon^2} \ln n$.

According to the Chernoff bound, if $Y_1 \ldots Y_z$ are independent random variables, $0 \leq Y_i \leq 1$, $Y = \Sigma_{i=1}^{z} Y_i$ and $E(Y) = \Sigma_{i=1}^{z} E(Y_i)$, then for any $\epsilon > 0$, $\Pr(|Y - E(Y)| \geq \epsilon E(Y)) \leq 2e^{-\frac{\epsilon^2}{2}E(Y)}$.

Applying the bound we get that $\Pr(X \leq (1-\epsilon)(\frac{4}{\epsilon^2}\ln n)) = \Pr(\Sigma_{i=0}^{\frac{4}{\epsilon^2}2^{2d-1}\ln n} X_i \frac{\epsilon^2}{4\ln n} \leq (1-\epsilon)\beta) \leq e^{-\frac{\epsilon^2}{2}\frac{4}{\epsilon^2}\ln n} = \frac{1}{n^2}$. When we do the same for less than n shifts, the overall probability that there exists a shift where the error is more than $\epsilon\beta$ is bounded by $\frac{1}{n}$. □

We now conclude with the main theorem for this subsection:

Theorem 7. *There exists a Monte Carlo randomized algorithm that solves the* Set Intersection Matching *problem with high probability in* $\Theta\left(4^d n \log n \log m\right)$ *time. The algorithm can also maintain a* $(1+\epsilon)$ *approximation to the mismatch count in time* $\Theta\left(\frac{1}{\epsilon^2}4^d n \log n \log m\right)$.

Proof. Applying Lemmas 5 and 6 provides the desired result. □

3.2. Deterministic algorithm with mismatch count

In this subsection we present a deterministic algorithm for the problem that can also count the number of mismatches between the pattern and any location in the text.

We define (S,i)-*match-count* for a set S of letters from Σ and index i in the text to be the number of times S is in the intersection, when checking for intersection match at index i of the text. Formally:

Definition 8. (S,i)-*match-count:* for $S \subseteq \Sigma$ and $0 < i \leq n-m$, (S,i)-match-count $= \left|\left\{j \mid 0 \leq j < m \text{ and } S \subseteq t_{i+j} \cap p_j\right\}\right|$.

We denote (S,i)-*match-count* by $\Phi(S,i)$. Consider the case where one is trying to match the pattern P to the text T at location i. We now show that:

Lemma 9. *The number of set intersection matches at location* i *is equal to*

$$\sum_{S\subseteq\Sigma, 0<|S|\leq d} (-1)^{|S|+1}\,\Phi(S,i) \tag{1}$$

Proof. First, we need the following result from the binomial expansion: by setting $x=-1$ in the binomial expansion $(1+x)^k = \sum_{l=0}^{k}\binom{k}{l}x^l$ we get

$$\begin{aligned} 0 &= \sum_{l=0}^{k}\binom{k}{l}(-1)^l \\ \Rightarrow -\binom{k}{0}(-1)^0 &= \sum_{l=1}^{k}\binom{k}{l}(-1)^l \\ \Rightarrow -1 &= \sum_{l=1}^{k}\binom{k}{l}(-1)^l \\ \Rightarrow 1 &= \sum_{l=1}^{k}\binom{k}{l}(-1)^{l+1} \end{aligned} \tag{2}$$

The value of formula (1) is the summation of all (S,i)-match-counts for all subsets S of odd size of Σ, minus all (S,i)-match-count for all subsets S of even size. Thus, for each of the locations we intersect and test for match (e.g. t_i against p_0, t_{i+1} against p_1, etc.). The formula counts the number of odd subsets of the intersection minus the number of even subsets and then sums these values.

Assuming the intersection size in such location is k, we get $\sum_{i=1}^{k}\binom{k}{i}(-1)^{i+1}$, which equals 1 using (2). Notice that for locations where the intersection is empty, the formula counts nothing. Hence, we see that formula (1) equals exactly the number of set intersection matches at location i as claimed by the lemma. □

The main theorem is:

Theorem 10. *The* Set Intersection Matching *problem can be solved and provide a mismatch count in* $\Theta\left(\sum_{i=1}^{d}\binom{\sigma}{i}n\log m\right) = O\left(\sigma^d n \log m\right)$ *time, deterministically.*

Table 3
Set intersection match count example.

S	$(S,0)$-match-count	$(S,1)$-match-count	$(S,2)$-match-count
{a}	1	1	2
{b}	1	1	0
{c}	1	1	1
{d}	1	1	0
{a,b}	1	0	0
{a,c}	0	0	1
{a,d}	0	0	0
{b,c}	0	0	0
{b,d}	0	0	0
{c,d}	0	1	0
sum(odd)–sum(even)	3	3	2

Table 4
Set intersection match count analysis.

S	Intersection (j)			$(S,3)$-match-count
	$\{a,d\}\cap\{a,b\}=\{a\}$	$\{a,c,d\}\cap\{a,c\}=\{a,c\}$	$\{b\}\cap\{c,d\}=\emptyset$	
{a}	1	1	0	2
{b}	0	0	0	0
{c}	0	1	0	1
{d}	0	0	0	0
{a,b}	0	0	0	0
{a,c}	0	1	0	1
{a,d}	0	0	0	0
{b,c}	0	0	0	0
{b,d}	0	0	0	0
{c,d}	0	0	0	0
sum(odd)–sum(even)	1	$2-1=1$	0	$3-1=2$

Proof. The value of $\Phi(S,i)$ for a given S subset of Σ can be calculated in $\Theta(n\log m)$ for all i's in the following way: First, we replace each set in the text and the pattern with 1 if S is a subset; it is 0 otherwise. We then use the convolution method to get the desired result.

Using Lemma 9, the proof of the theorem is now trivial. We already see that $\Phi(S,i)$ can be calculated in time $\Theta(n\log m)$, and the number of elements summed up in the lemma is exactly $\sum_{i=1}^{d}\binom{\sigma}{i}$. (Note that $d\leq\sigma$, as all sets are subsets of Σ.) □

Example 3. Consider the text $T=$ [{a,b}, {b,c,d}, {a,d}, {a,c,d}, {b}] and the pattern $P=$ [{a,b}, {a,c}, {c,d}]. The (S,i)-match-count for all possible indexes i and for all relevant subsets $S\subseteq\Sigma$ are described in Table 3. One can see that the result is exactly the mismatch count of the *Set Intersection Matching* problem for each matching location i.

Example 4. Consider the case where one is trying to calculate the match count between the above pattern to the above text at location 2 of the text. Analysis of correctness of the algorithm for that case is described in Table 4. The last column, which sums up the rows, is exactly the same as the last column of the previous example. By summing up columns (odd size–even size) we get 1 for each index where the intersection is not empty, and 0 otherwise. Hence, it is clear why we get the match count (i.e., 2) when summing up the last column (odd size–even size).

4. Sequence Matching algorithm

In this section we first show a simple algorithm for the *Sequence Matching* problem in the case $d=2$, and then generalize it for $d>2$.

4.1. Simple case study ($d=2$)

Assuming the length of all sequences is 1 or 2, it is easy to see that one can pad all the sequences that are smaller than 2 with unused letters (one for text sequences, and another one for pattern sequences), to get all sequences to be of the same length. Thus, we can assume that all sequences are of length 2.

Theorem 11. *The* Sequence Matching *problem, in the case of* $d=2$ *can be solved in* $\Theta(n\log m)$ *time, deterministically.*

Proof. Denote the sequences of text $T = t_0, t_1, t_2, \ldots, t_{n-1}$ by $t_i = [x_i, y_i]$, and the sequences of the pattern $P = p_0, p_1, p_2, \ldots, p_{m-1}$ by $p_i = [a_i, b_i]$ while minding the following formula for a pre-defined i:

$$\sum_{j=0}^{m-1} \left[\left(x_{i+j} - a_j\right) \left(y_{i+j} - b_j\right) \right] \tag{3}$$

Each of the summands in (3) equals 0 if and only if $x_{i+j} = a_j$ or $y_{i+j} = b_j$, which means the two sequences match. It thus equals 0 if the text matches the pattern at index i.

Expanding each summand yields:

$$\sum_{j=0}^{m-1} \left(x_{i+j} y_{i+j} - a_j y_{i+j} - b_j x_{i+j} + a_j b_j\right) \tag{4}$$

To solve the problem one should get the value of the formula for all $0 \leq i \leq n - m$. This can be done by calculating each of the four parts independently, and summing them afterwards. The first and last parts can be easily calculated for all i's in $\Theta(n + m)$ time. This is done with one pass over the text and pattern and handling a window. The two other parts can be easily calculated for all i's as well in $\Theta(n \log m)$ time using convolutions [4].

Notice that the sum for few i's can be 0 by chance even in cases where the pattern does not match the text. This happens when the sum of then negative summands equals the positive summands. To avoid that, one should calculate the sum of the squares in the formula. More details are given in the next subsection. □

4.2. The general solution

We now generalize the previous algorithm for cases where $d > 2$. As in the former case, it is easy to see that one can pad all the sequences that are smaller than d with unused letters, in order to get all sequences to be of the same length. Thus, we can assume that all sequences are of size d. The following theorem with its proof shows that there is an algorithm for solving the problem in $\Theta\left(4^d n \log m\right)$:

Theorem 12. *The* Sequence Matching *problem can be solved in* $\Theta\left(4^d n \log m\right)$ *time.*

Proof. Denote the sequences of the text and pattern by $t_i = [x_{i_0}, x_{i_1} \ldots x_{i_{d-1}}]$ and $p_i = [a_{i_0}, a_{i_1} \ldots a_{i_{d-1}}]$, respectively, and consider the following formula for a pre-defined i:

$$\sum_{j=0}^{m-1} \left[\left(x_{i+j_0} - a_{j_0}\right) \left(x_{i+j_1} - a_{j_1}\right) \ldots \left(x_{i+j_{d-1}} - a_{i_{d-1}}\right) \right] \tag{5}$$

As before, each summand in (5) equals 0 if and only if $x_{i+j_0} = a_{j_0}$ or $x_{i+j_1} = a_1$ or … or $x_{i+j_{d-1}} = a_{j_{d-1}}$. This means that the two sequences match. Hence the sum of the formula, which is the result of summation of m contiguous such parts, equals 0 if there is a match at index i.

Expanding each summand yields:

$$\sum_{j=0}^{m-1} \left(x_{i+j_0} x_{i+j_1} \ldots x_{i+j_{d-1}} - x_{i+j_0} x_{i+j_1} \ldots x_{i+j_{d-2}} a_{j_{d-1}} \ldots + (-1)^d a_{j_0} a_{j_1} \ldots a_{j_{d-1}} \right) \tag{6}$$

To solve the problem one should get the value of the formula for all $0 \leq i \leq n - m$. This can be done by calculating each term independently and summing it later. Notice that the formula consists of 2^d terms, where each of them is multiplication of the text elements of a specific sequence as well as pattern elements of another sequence. For each part, the multiplication between elements from the text can be calculated for all i's in linear time. The same can be done for multiplication between elements from the pattern and using a window as in the previous subsection. Using convolutions we can then get the term's value for all $0 \leq i \leq n - m$, in total time $\Theta\left(2^d n \log m\right)$.

As before, to avoid getting 0 by chance, one should calculate the sum of squares of the formula. All parts of formula (5) will be squared, and the result is $2^{2d} = 4^d$ parts in the extracted formula with similar structure. □

References

[1] R. Cole, R. Hariharan, Tree pattern matching to subset matching in linear time, SIAM J. Comput. 32 (4) (2003) 1056–1066.

[2] R. Cole, R. Hariharan, P. Indyk, Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time, in: Proc. 10th ACM–SIAM Symposium on Discrete Algorithms, SODA, 1999, pp. 245–254.

[3] Amihood Amir, Ely Porat, Moshe Lewenstein, Approximate subset matching with Don't Cares, in: Proceedings of the Twelfth Annual ACM–SIAM Symposium on Discrete Algorithms, January 07–09, 2001, Washington, D.C., United States, 2001, pp. 305–306.

[4] M.J. Fischer, M.S. Paterson, String-matching and other products, Technical Report, UMI Order Number: TM-41, Massachusetts Institute of Technology, 1974.
[5] Raphael Clifford, Klim Efremenko, Ely Porat, Amir Rothschild, From coding theory to efficient pattern matching, in: ACM–SIAM Symposium on Discrete Algorithms, SODA, 2009, pp. 778–784.
[6] Tobias Marschall, Sven Rahmann, Probabilistic arithmetic automata and their application to pattern matching statistics, in: CPM, 2008, pp. 95–106.
[7] D.E. Knuth, J.H. Morris, V.B. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.
[8] P. Clifford, R. Clifford, Simple deterministic wildcard matching, Inform. Process. Lett. 101 (2) (Jan. 2007) 53–54.
[9] R. Cole, R. Hariharan, Verifying candidate matches in sparse and wildcard matching, in: Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, May 19–21, 2002, STOC '02, ACM, New York, NY, 2002, pp. 592–601.
[10] Zvi Galil, On improving the worst case running time of the Boyer–Moore string matching algorithm, Commun. ACM 22 (9) (September 1979) 505–508.
[11] Amihood Amir, Moshe Lewenstein, Ely Porat, Faster algorithms for string matching with k mismatches, in: Proceedings of the Eleventh Annual ACM–SIAM Symposium on Discrete Algorithms, SODA '00, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, pp. 794–803.
[12] P. Clifford, R. Clifford, C.S. Iliopoulos, Faster algorithms for δ, γ-matching and related problems, in: A. Apostolico, M. Crochemore, K. Park (Eds.), CPM 2005, in: LNCS, vol. 3537, Springer, Heidelberg, 2005, pp. 68–78.
[13] A. Amir, O. Lipsky, E. Porat, J. Umanski, Approximate matching in the L_1 metric, in: A. Apostolico, M. Crochemore, K. Park (Eds.), CPM 2005, in: LNCS, vol. 3537, Springer, Heidelberg, 2005, pp. 91–103.
[14] P. Indyk, Faster algorithms for string matching problems: matching the convolution bound, in: Proceedings of the 39th IEEE Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, 1998, pp. 166–173.