Experimental (δ, γ)-Pattern-Matching with Don't Cares

Costas S. Iliopoulos*, Manal Mohamed*, Velumailum Mohanaraj*
*King's College London/Department of Computer Science, London WC2R 2LS, UK

***Abstract*— The δ-Matching problem calculates, for a given text $T_{1..n}$ and a pattern $P_{1..m}$ on an alphabet of integers, the list of all indices $\mathcal{I}_\delta = \{i : \max_{j=1}^{m} |P_j - T_{i+j-1}| \leq \delta\}$. The γ-Matching problem computes, for given T and P, the list of all indices $\mathcal{I}_\gamma = \{i : \sum_{j=1}^{m} |P_j - T_{i+j-1}| \leq \gamma\}$. When a *"don't care"* symbol occurs, the associated difference is counted as zero. In this paper, we give experimental results for the different matching algorithms that handle the presence of "don't care" symbols. We highlight some practical issues and present experimental analysis for the current most efficient algorithms that calculate $\mathcal{I}_\delta, \mathcal{I}_\gamma$, and $\mathcal{I}_{(\delta,\gamma)} = \mathcal{I}_\delta \cap \mathcal{I}_\gamma$, for pattern P with occurrences of "don't cares". Moreover, we present our own version of γ-Matching algorithm.**

Keywords: δ-matching, γ-matching, wildcard matching, music information retrieval.

I. Introduction

The string pattern-matching problem is to find all the occurrences of a given pattern $P_{1..m}$ as a substring of a larger text $T_{1..n}$, both being sequences of symbols (characters) drawn from a finite alphabet Σ. This problem has plethora of applications, such as text retrieval, musical analysis, computational biology, image analysis, data mining and network security.

In this paper we focus on a set of pattern-matching problems that arise in computer assisted musical analysis. A musical score can be viewed as a string over an alphabet consisting of integers, representing the set of notes in chromatic or diatonic notion, or the set of intervals that appear between notes (*e.g.* chromatic pitch may be represented as MIDI numbers in the range 0 through 127). In spite of such representation, existing efficient algorithms in the field of string pattern-matching, specifically exact matching, could not be used to solve some problems related to musical analysis. For instance, it could not be used for locating melodies within songs. This is because two songs of the same melody are usually made of "similar" notes, but rarely exactly the same.

Throughout the years, many measures to account similarity between closely related but non-identical musical strings have been developed. Several of such depend on allowing some tolerance in the matching process. For example, the δ-Matching allows a difference of at most δ units between each symbol in the pattern and the corresponding symbol in the text. Also, the γ-Matching bounds the sum of the absolute differences between the corresponding values of the pattern and the text by γ. The combination of these two approximates is commonly known as the (δ, γ)-Matching. (Similar measures have been used in image template matching [3].)

Very recently, the δ-, γ-, and (δ, γ)-Matching were extended to completely ignore the differences corresponding to some positions in the pattern during the matching process. This new variant of the problem reports the approximate occurrences of the pattern in the text, even if some particular notes in the pattern are played slightly out of tune or recorded incorrectly. This special matching problem is seen as having "don't care"[1] symbols in the pattern, where a "don't care" symbol is a symbol that matches every other symbol in the alphabet including itself.

Fast Fourier transforms (FFT) have been the basis for designing efficient algorithms for several approximate string-matching problems. For instance, Cole and Hariharan [7] presented an $O(n \log m)$-time algorithm for the exact pattern-matching problem with "don't cares". Additionally, Clifford *et. al.* [6] devised an $O(\delta n \log m)$-time algorithm for the δ-Matching problem. Using same approach, an $O(\delta n \log m)$ algorithm was given for the (δ, γ)-Matching, which broke the then known bound of $O(mn)$ for the problem when δ is $o(m/\log m)$. Furthermore, Clifford *et. al.* [6] provided a faster algorithm for the γ-Matching problem, which runs in $O(n\sqrt{m \log m})$ time and is based on a divide and conquer approach. Independently, Amir *et. al.* [1] also provided an algorithm for γ-Matching that has the same time complexity as [6]. Another efficient algorithm presented by Lipsky [13] for the δ-Matching has running time of $O(|\Sigma|\, n(\log m + |\Sigma|))$.

Ardila *et. al.* [2] provided different algorithms for handling the presence of "don't cares" in the pattern for the δ-, γ- and (δ, γ)-Matching problems, by extending the then best algorithms for these problems. In the context of δ-, γ- and (δ, γ)-Matching, the "don't care" symbols when aligned with any symbols in the alphabet do not contribute any values towards the distances. To be more precise, the distance between the "don't care" symbol and any symbols in the alphabet or that symbol itself is zero. The algorithms extended in [2] are due to [1], [6], [13] mentioned above, and have the same running time as their original versions.

This paper presents experimental results from an empirical study which was done by implementing all the algorithms in [2], [6]. These results will help one to choose the

[1] Also known as wild card symbol.

right algorithm for a problem in hand from the variants of algorithms available. The γ-Matching algorithm presented in [6] fails in some cases. We provide solutions for these problems. We also present a new $O(|\Sigma| n \log m)$-time algorithm for the γ-Matching problem. This method is asymptotically faster than the $O(n\sqrt{m \log m})$-time algorithm due to [6], when $|\Sigma|$ is $o(m/\log m)$. A way of improving the δ-Matching algorithm due to [2] is also given.

The outline of the paper is as follows: Some preliminaries are described in Section II. In Section III we present the two δ-Matching algorithm and suggest ways for improving their performance. A γ-Matching algorithm that runs in $O(|\Sigma| n \log m)$ and a rectification for an existing γ-Matching algorithm are given in Section IV. In Section 5, we briefly outline the (δ, γ)-Matching algorithm. All the algorithms presented in the paper handle the occurrence of "don't cares" in the pattern. The experimental results are presented in Section VI. Finally, we conclude in Section VII.

II. Preliminaries

Throughout the paper, the *alphabet* Σ is assumed to be an interval of integers and considered to be $\Sigma = \{1, 2, ..., |\Sigma|\}$. A *text* $T = T_{1..n}$ is a string of length n defined on Σ. T_i is used to denote the *i-th element* of T, and $T_{i..j}$ is used as a notation for the *substring* $T_i T_{i+1} \cdots T_j$ of T, where $1 \leq i, j \leq n$. Similarly, a *pattern* $P = P_{1..m}$ is a string of length m defined on $\Sigma \cup \{\star\}$, where $\star$ is the *don't care* symbol.

The "don't care" *matches* every symbol including itself, that is, $\star = a$ for each $a \in \Sigma \cup \{\star\}$. A pattern P is said to *occur* in T at position i if $P_j = T_{i+j-1}$, for $1 \leq j \leq m$.

Let δ, γ be two given integers, then two patterns P^1 and P^2 are said to be *δ-matched* (denoted as $P^1 =_\delta P^2$), if $\max_{j=1}^{m} |P^1_j - P^2_j| \leq \delta$. Additionally, P^1 and P^2 are said to be *γ-matched* (denoted as $P^1 =_\gamma P^2$), if $\sum_{j=1}^{m} |P^1_j - P^2_j| \leq \gamma$, where $|\star - a| = |a - \star| = 0, \ \forall\ a \in \Sigma \cup \{\star\}$. The δ, γ and (δ, γ)-Matching problems are defined as follows:

Definition 1 (δ-Matching Problem)**.** *For a given text T, pattern P and integer δ, the δ-Matching (also known as L_∞-Matching) problem is to calculate the set of all indices, $\mathcal{I}_\delta$, such that if $T_{i..i+m-1} =_\delta P$, then $i \in \mathcal{I}_\delta$. In other words, $\mathcal{I}_\delta = \{i : \max_{j=1}^{m} |P_j - T_{i+j-1}| \leq \delta\}$.*

Definition 2 (γ-Matching Problem)**.** *For a given text T, pattern P and integer γ, the γ-Matching (also known as L_1-Matching) problem is to calculate the set of all indices, $\mathcal{I}_\gamma$, such that if $T_{i..i+m-1} =_\gamma P$, then $i \in \mathcal{I}_\gamma$. In other words, $\mathcal{I}_\delta = \{i : \sum_{j=1}^{m} |P_j - T_{i+j-1}| \leq \gamma\}$.*

Definition 3 ((δ, γ)-Matching Problem)**.** *For a given text T, pattern P and integers δ, γ; the (δ, γ)-Matching problem is to calculate the set of all indices, $\mathcal{I}_{(\delta,\gamma)}$, such that: $\forall\ i \in \mathcal{I}_{(\delta,\gamma)}$,*

(1) $T_{i..i+m-1} =_\delta P$, and

(2) $T_{i..i+m-1} =_\gamma P$.

In this paper, we will assume the standard RAM model of computation, which allow arithmetic on $\log N$ bit numbers in $O(1)$ time, where N is of the order of the maximum problem size. For such model, the following theorem is standard and crucial to our algorithms.

Theorem 1. *Consider two numerical strings $X_{1..n}$ and $Y_{1..m}$. Then, the* convolution *$Z_{1..n} = X \otimes Y$ can be computed accurately and efficiently in $O(n \log m)$ time using Fast Fourier Transform (FFT).*

The above theorem holds because the Fourier transform converts convolution into elementwise multiplication. The theorem presents one of the early breakthroughs in algorithms with considerable numbers of applications (see [9], [11] and [12]).

III. δ-Matching with Don't Cares Algorithm

Informally, the δ-Matching problem computes all indices i such that the maximum $|P_j - T_{i+j-1}|$ over all j's is no larger than δ. In the following subsections, we briefly present two algorithms for the δ-Matching problem in the presence of "don't cares" (Both algorithms were presented in [2].). When necessary, we will highlight some practical issues and suggest practical solutions.

A. An $O(|\Sigma| n(\log m + |\Sigma|))$-Time Algorithm

Original Algorithm. Based on ideas used in [13] to compute L_∞-matching for strings without "don't cares", Ardila *et. al.* [2] presented an algorithm for the δ-Matching with "don't cares". The idea is to encode the text and the pattern in such way that one convolution, and linear time pass on the convolution's output are sufficient to compute the desired output. The main steps of the algorithm are as follows:

Step 1: Encode both the text T and the pattern P in such way that every symbol $\sigma \in \Sigma \cup \{\star\}$ is represented by a binary string, over $\{0, 1\}^*$, of length $2|\Sigma|$. In particular, $\sigma = T_i$ $(1 \leq i \leq n)$ will be replaced by the sequence $c^t(\sigma) = c^t(\sigma)_1, \ldots, c^t(\sigma)_{2|\Sigma|}$, where

$$c^t(\sigma)_j = \begin{cases} 1, & \text{if } j = |\Sigma| + \sigma \\ 0, & \text{otherwise} \end{cases} \quad \text{for } 1 \leq j \leq 2|\Sigma|.$$

Additionally, every symbol $\sigma = P_i$ $(1 \leq i \leq m)$ will be replaced by

$$c^p(\sigma)_j = \begin{cases} 1, & \text{if } j = \sigma \text{ and } \sigma \neq \star \\ 0, & \text{otherwise} \end{cases} \quad \text{for } 1 \leq j \leq 2|\Sigma|.$$

Step 2: Here, the algorithm performs the convolution $R = c^t(T) \otimes c^p(P)$.

Step 3: Finally, we construct a string $D_{1..n}$, over $\{0, 1\}^*$, such that D_i $(1 \leq i \leq n - m + 1)$ will be 1, if and only if, P occurs at position i of T; otherwise it will be 0. The values of D are obtained as follows:

$$D_i = \begin{cases} 1, & \text{if } \sum_{j=q-\delta}^{q+\delta} R[j] = m' \\ 0, & \text{otherwise} \end{cases} \quad 1 \leq i \leq n - m + 1,$$

where $q = (2(i-1)+1)|\Sigma|+1$, and m' is the number of non-"don't cares" in P.

In this way the algorithm computes the desired results, $\mathcal{I}_\delta = \{i : D_i = 1\}$. This concludes the algorithm.

Known Issues. The above algorithm is based on the assumption that the alphabet is $\Sigma = \{1, 2, \ldots |\Sigma|\}$. This assumption gives rise to two issues:

1) In practice, 0 is a valid numerical representation of musical notes. For instance, lowest note upon a MIDI controller is a C and this note is assigned note number 0. Moreover, when a melody is represented as a string of pitch intervals, the string might include negative numbers.
2) The real musical representation might have all notes in a small range, but not starting from 1. For example, consider C-minor 7 sequence $\{60, 63, 67, 70\}$. If the $|\Sigma|$ is taken as 70, the algorithm gives the correct result, but would be rather slow. Because, each element will be encoded to a binary string of length 140.

Solutions. By modifying the encoding scheme, the above issues can be addressed efficiently. The idea is to map the elements in the range $\{a, \ldots, b\}$ to another but equal range, $\{1, \ldots, b-a+1\}$. Applying such technique to the above example, $\{60, 63, 67, 70\}$, yields $\{1, 4, 8, 11\}$. Since both pattern and text elements are mapped this way, the relative difference will not be affected.

B. An $O(\delta n \log m)$-Time Algorithm

Original Algorithm. In [6], an $O(\delta n \log m)$-time algorithm for the δ-Matching problem without "don't cares" is presented. A function that is zero when there is a match between two symbols and bounded away from zero otherwise, is constructed. Standard properties of the even periodic functions and their discrete cosine transform [9] are used to achieve such goals. An even periodic function $f_\delta(x)$ that is equal to x^2 for $|x| \leq \delta$ is used to construct the desired function as follows:

$$d(x-y) = (x-y)^2 - f_\delta(x-y).$$

Note that $d(x-y) = 0$ if $|x-y| \leq \delta$, and $d(x-y) > 0$ otherwise. Thus, to perform δ-Matching we need to compute $\sum_{j=1}^{m} d(P_j - T_{i+j-1})$, for $1 \leq i \leq n-m+1$. Calculating each $d(P_j - T_{i+j-1})$ involves $O(\delta)$ inner products and requires a total of $O(\delta n \log m)$ running time; see [6] for details.

Ardila *et. al.* [2] extended the above algorithm to handle the occurrences of the "don't cares" in the pattern as follows:

Step 1: If $\mathcal{J}^\star = \{j \mid P_j = \star\}$. Then, a new pattern $P^{(0)}$ is created as follows:

$$P_j^{(0)} = \begin{cases} 0, & \text{if } j \in \mathcal{J}^\star; \\ P_j, & \text{otherwise.} \end{cases}$$

Additionally, the values $\mathcal{D}_i^1 = \sum_{j=1}^{m} d(P_j^{(0)} - T_{i+j-1})$, for $1 \leq i \leq n-m+1$ are computed using the δ-Matching algorithm in [6].

Step 2: The aim of this step is to compute for each position i in T the value $\sum_{j \in \mathcal{J}^\star} d(T_{i+j-1})$. To achieve this, a new pattern $P^{(1)}$ is computed as follow:

$$P_j^{(1)} = \begin{cases} 1, & \text{if } j \in \mathcal{J}^\star; \\ 0, & \text{otherwise.} \end{cases}$$

Additionally, a new text T' is computed as follows: $T'_i = d(T_i) = T_i^2 - f_\delta(T_i)$. Once $P^{(1)}$ and T' are obtained, the values $\sum_{j \in \mathcal{J}^\star} d(T_{i+j-1})$, for $1 \leq i \leq n-m+1$, can be easily obtained as $\mathcal{D}^2 = T' \otimes P^{(1)}$.

Step 3: In this step the actual δ-Matching is computed as follows:

$$\mathcal{D}_i = \mathcal{D}_i^1 - \mathcal{D}_i^2.$$

Observe that $\mathcal{I}_\delta = \{i : D_i = 0\}$. This concludes the algorithm.

IV. γ-Matching with Don't Cares Algorithm

Informally, the γ-Matching problem computes all indices i such that the sum of differences $|P_j - T_{i+j-1}|$ over all j's is no larger than γ. In the following subsections, we present two algorithms for the γ-Matching problem in the presence of "don't cares". The first algorithm was presented in [2]. During the implementation, the algorithm failed to calculate the desired output. We will briefly explain the algorithm and provide corrections.

A. The $O(n\sqrt{m \log m})$-Time Algorithm

Using the divide and conquer approach, Clifford *et. al.* [6] presented an $O(n\sqrt{m \log m})$-time algorithm for the γ-Matching problem. This algorithm was extended by Ardila *et. al.* to handle the occurrences of "don't cares" in the pattern; see [2] for details. We found some problems with the original γ-Matching algorithm. Therefore, we will elaborate on each step of the algorithm. Then, we will explain the arisen problems and the proposed solutions.

Original Algorithm. The γ-Matching algorithm is based on the following observation: Given two numbers x, and y and an arbitrary value θ, if it is known that x and y are on the opposite side of θ, the absolute value $|x-y|$ can be calculated as the sum of four products

$$|x-y| = xI_xJ_y - I_xyJ_y + J_xyI_y - xJ_xI_y, \quad (1)$$

where

$$I_a = \begin{cases} 1, & \text{if } a > \theta \\ 0, & \text{otherwise;} \end{cases} \quad \text{and} \quad J_a = \begin{cases} 1, & \text{if } a \leq \theta \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Using a number of θ values, namely $\theta_1, \ldots, \theta_b$ rather than using a single θ, a divide-and-conquer strategy is applied to the γ-Matching problem as follows:

Step 1: Calculate the integer $b = \lceil \sqrt{m/logm} \rceil$.

Step 2: Sort the values in $P_{1..m}$ and let A be the associated array of indices of these values. Partition A into b successive arrays denoted by $\{A_1, A_2 \dots A_b\}$, each having a length of $\lceil m/b \rceil$. Define θ values in this way: $\theta_0 = -\infty$ (an impossibly small value in p or t), and $\theta_k = max\{p_j : j \in A_k\}$ for $k = 1, \dots, b$.

Step 3: Sort the elements in T and divide them into b groups as shown below. Let B_k be the array of indices of k-th group.

$\theta_0 < T_i \leq \theta_1$	$\theta_1 < T_i \leq \theta_2$	$\dots$ $\theta_{b-2} < T_i \leq \theta_{b-1}$	$\theta_{b-1} < T_i$

Step 4: For each θ_k $(1 \leq k \leq b)$, define I_k, and J_k functions (compare with (2))

$$I_k(x) = \begin{cases} 1, & \text{if } x > \theta_k \\ 0, & \text{otherwise} \end{cases}$$

and

$$J_k(x) = \begin{cases} 1, & \text{if } \theta_{k-1} < x \leq \theta_k \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

Additionally, for each k, create four new text strings $\epsilon_{k1}(T), \epsilon_{k2}(T), \epsilon_{k3}(T)$ and $\epsilon_{k4}(T)$ with j-th elements that are respectively $T_j I_k(T_j), I_k(T_j), J_k(T_j)$ and $T_j J_k(T_j)$ for each $k = 1, \dots, b$. Correspondingly, create four new pattern strings $\upsilon_{k1}(P), \upsilon_{k2}(P), \upsilon_{k3}(P)$, and $\upsilon_{k4}(P)$ that are respectively $J_k(P_j), -P_j J_k(P_j), P_j I_k(P_j)$ and $-I_k(P_j)$.

The idea is straight forward now: the total difference can be calculated using b steps, each includes 4 summations as follows:

$$\sum_{j=1}^{m} |P_j - T_{i+j-1}| = \sum_{k=1}^{b} \sum_{l=1}^{4} \upsilon_{kl}(P)\epsilon_{kl}(T) + \textit{remainder},$$

where the remainder will be calculated in the next step.

Step 5: Here we have to deal with the cases ignored in Step 4. Recall that (1) calculates the difference only when the two values are on opposite sides of the threshold θ. Therefore, the differences of the pairs (P_j, T_{i+j-1}) where $P_j \in A_k$ and $T_{i+j-1} \in B_k$ for some k would have been omitted, as the pairs of text and pattern elements in the associated arrays (A_k, B_k) lie on the same side to all thresholds θ_k $(1 \leq k \leq b)$. This omitted differences have to be computed separately and added to the corresponded total difference. This can be done in a straightforward manner, see [6] for details.

Known Issues. The above algorithm does not seem to work as stated. This is mainly due to the asymmetrical behavior of Equation (1) when x or y coincides with θ. This is better explained with two examples.

Example 1. *If $\theta = 20, x = 18$ and $y = 20$, then calculating $|x - y|$ using the Equation (1) incorrectly gives 0. Here, θ equals the larger number.*

Example 2. *If $\theta = 20, x = 20$ and $y = 22$, then calculating $|x - y|$ using Equation (1) correctly gives 2. Here, θ equals the smaller number.*

The examples clearly show that when θ equals the smaller of x and y, the difference is calculated correctly and when θ equals the larger of x and y it is evaluated as zero irrespective of the actual difference.

Recall that the main idea of the algorithm is to accumulate the contributions to the total difference from pairs of values (P_j, T_{i+j-1}) on opposite sides of specific thresholds first, and then collect the remaining terms in the second stage of the algorithm. In the following we explain the two main problems in the algorithm.

1) Consider iteration k where θ_k is used as the threshold and the differences between the pattern elements $\{P_j \mid j \in A_k\}$ and the text elements $\{T_j \mid j \in \{B_{k+1} \cup \dots B_b\}\}$, and the differences between the text elements $\{T_j \mid j \in B_k\}$ and the pattern elements $\{P_j \mid j \in \{A_{k+1} \cup \dots A_b\}\}$ are accumulated.
   Let's consider the differences between the elements in the partitions (A_{k+1}, B_k) among other differences computed at the k-th iteration. For the differences to be accumulated during this iteration, the elements in these partitions must be on the opposite side of θ_k. But this may not be the case for some elements. There may be elements in A_{k+1} that are equal to θ_k. At the same time, we know for certain that the elements in B_k are less or equal to θ_k. Therefore, pairs of values (P_j, T_{i+j-1}) where $\{P_j \mid P_j = \theta_k \text{ and } j \in A_{k+1}\}$ and $\{T_{i+j-1} \mid T_{i+j-1} < \theta_k \text{ and } (i+j-1) \in B_k\}$ will not be on the opposite side of θ_k, rather it would be like the case shown in *Example 1* above and will be omitted in Step 4. They are not handled in Step 5 too. Therefore, at the end of the algorithm, the total difference might have smaller values than the actual values at some positions.
2) Consider the partitions A_k and B_k for some $1 \leq k \leq b$. By definition, the block A_k contains the indices j of the pattern elements P_j such that $\theta_{k-1} \leq P_j \leq \theta_k$, and the block B_k contains all the indices j of the text elements T_j such that $\theta_{k-1} < T_j \leq \theta_k$ for $1 \leq k < b$ and B_b contains all the indices j of the text elements T_j such that $T_j > \theta_{b-1}$. Thus, there may be elements belonging to A_k, which are equal to θ_{k-1} and there may be elements belonging to B_k, which are greater than θ_{k-1}. When θ_{k-1} is used as the threshold, by what we established earlier using the Example 2, the difference between these elements, i.e., $\{P_j \mid P_j = \theta_{k-1} \text{ and } j \in A_k\}$ and $\{T_j \mid t_j > \theta_{k-1} \text{ and } j \in B_k\}$, will be computed correctly and added to the total difference in Step 4 of the algorithm.
   In spite of this, the next step, i.e., Step 5, of the algorithm makes a wrong assumption that *all* pair of text and pattern elements that lie in associated arrays (A_k, B_k) for some k would have been omitted in Step 4 and computes and adds them again to the

total difference. However, this assumption does not hold for some elements. So the difference between the elements $\{P_j \mid P_j = \theta_{k-1} \text{ and } j \in A_k\}$ and the those elements $\{T_j \mid T_j > \theta_{k-1} \text{ and } j \in B_k\}$ are added to the total difference exactly *twice*.

Solutions. The first problem can be addressed by modifying Step 4 of the algorithm as follows: For each θ_k $(1 \le k \le b-1)$, define different I, and J functions (compare with (3))

$$I_k(x) = \begin{cases} 1, & \text{if } x \ge \theta_k \\ 0, & \text{otherwise,} \end{cases}$$

and

$$J_k(x) = \begin{cases} 1, & \text{if } \theta_{k-1} < x \le \theta_k \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Due to this modification, difference between some pairs of text and pattern elements that lie in associated arrays (A_k, B_k) for some k will be added to the total difference at Step 4 itself. To be precise, $\{P_j \mid P_j = \theta_k \text{ and } j \in A_k\}$ and $\{T_j \mid T_j < \theta_k \text{ and } j \in B_k\}$ will be added to the total difference at Step 4. This is quite similar to the second problem above and can be solved together by modifying Step 5 of the algorithm as follows:

Handling the omitted cases (Step 5)

Input: $\{A_1, A_2 \dots A_b\},\ \{B_1, B_2 \dots B_b\},\ b,\ P,\ T,\ M,\ \theta$
Output: M

1. **for** $k = 1$ **to** $b-1$ **do**
2. **for** each element $j \in B_k$ **do**
3. **for** each element $i \in A_k$ **do**
4. **if** $((j \ge i)$ **and** $(p_i \notin \{\theta_{k-1}, \theta_k\})$ **and** $(t_j \ne \theta_k))$ **then**
5. $M_{j-i+1} \leftarrow M_{j-i+1} + |p_i - t_j|$
6. **for** each element $j \in B_b$ **do**
7. **for** each element $i \in A_b$ **do**
8. **if** $((j \ge i)$ **and** $(p_i \ne \theta_{b-1}))$ **then**
9. $M_{j-i+1} \leftarrow M_{j-i+1} + |p_i - t_j|$

B. An $O(|\Sigma| n \log m)$-Time Algorithm

An even periodic function can be decomposed into a weighted sum of simpler trigonometric component functions. In this way, the function can be expressed in terms of $O(2 \times period)$ product terms. Then, FFTs can be used for the computation of these inner-products efficiently. This idea was used in [6] to devise faster algorithms for δ and (δ, γ)-Matching problems.

In the following, we explain how a different even periodic function can be used; in the same way, to efficiently solve the γ-Matching problem. The proposed function is as follow:

$$f(x) = |x| \quad for \quad |x| \le \ell,$$

where ℓ is the maximum absolute difference between any element in T and any element in P. The above function can be used to find the total difference as follows:

$$\sum_{j=1}^{m} f(p_j - t_{i+j-1}).$$

Recall that $f(x)$ can be expressed as a linear combination of $2\ell - 1$ product terms. Then, $f(x-y)$ can be expressed as follows:

$$f(x-y) = \alpha_0 r(0) + \sum_{k=1}^{\ell} \alpha_k r(k) c_k(x) c_k(y) + \sum_{k=1}^{\ell-1} \alpha_k r(k) s_k(x) s_k(y), \quad (5)$$

where

$$\alpha_k = \sum_{x=1-\ell}^{\ell} f(x) h_k(x), \qquad c_k(x) = \cos(xk\pi/\ell),$$
$$h_k(x) = r(k)\cos(xk\pi/\ell), \qquad s_k(x) = \sin(xk\pi/\ell),$$

and $r(k) = 1/\sqrt{2\ell}$ if $k \bmod \ell = 0$ and $r(k) = 1/\sqrt{\ell}$, otherwise.

Let's analyze our algorithm. the value ℓ van be calculated in linear time. Then the total difference can be computed; as in Equation (5), using $2\ell - 1$ inner products, each of which takes $O(n \log m)$ time. Computation of γ-Matching from the total difference can be performed in linear time. Therefore, the total running time is $O(\ell n \log m)$. Since $\ell = O(|\Sigma|)$, the overall running time is $O(|\Sigma| n \log m)$.

This algorithm can be extended as in Subsection III-B to handle the occurrence of "don't cares" in P. Thus, the γ-Matching for pattern with "don't cares" can be calculated in $O(|\Sigma| n \log m)$ time.

V. (δ, γ)-Matching with Don't Cares Algorithm

In [6], an $(\delta n \log m)$-time algorithm for (δ, γ)-Matching without "don't cares" was presented. The algorithm follows the same method for the δ-Matching algorithm. The only difference is the choice of the even periodic function. Moreover the algorithm was extended in [2] in the same way as described in Subsection III-B, to handle the occurrences of the "don't care" symbols in the pattern P in $O(\delta n \log m)$ time; see [2], [6] for details.

VI. Experimental Results

We implemented in C++, in homogeneous way, the following five algorithms: two alternative algorithms for the δ-Matching problem with "don't cares", the first runs in $O(|\Sigma|\, n(\log m + |\Sigma|))$ time, and the second requires $O(\delta n \log m)$ time, another two algorithms for γ-Matching problem with "don't cares", one is due to [2] and runs in $O(n\sqrt{m \log m})$ time and the other one is presented in this paper and requires $O(|\Sigma|\, n \log m)$ time, and an $O(\delta n \log m)$-time algorithm for (δ, γ)-Matching. In the rest of this section, these algorithms will be referred as *δ-Version 1*, *δ-Version 2*, *γ-Version 1*, *γ-Version 2* and *(δ, γ)-Version 1* respectively.

Remark 1. *The test data was extracted mainly from the albums* Candle In The Wind 1997 *by Elton John,* Baby One More Time *by Britney Spears,* Poovellam Un Vaasam *by Vidyasagar and* Ali Arjuna *by A. R. Rahman. The extracted data was tweaked, without destroying the*

melody, to change the alphabet size and their length for the experiments. The "don't cares" were placed in the patterns at random positions.

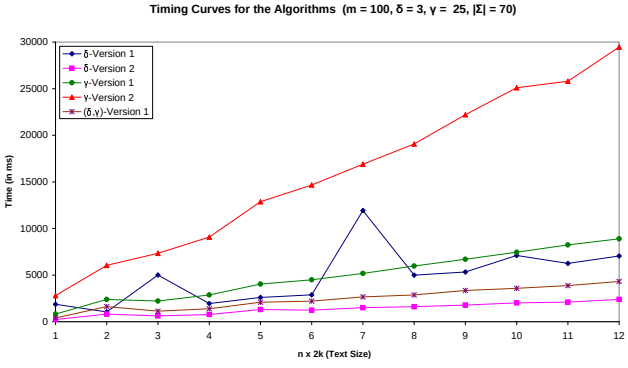


Fig. 1. Timing curves for δ-, γ-, (δ,γ)-Matching algorithms with "don't cares".

A. δ-Matching with Don't Cares

The two versions of the algorithms for this problem, namely *δ-Version 1* and *δ-Version 2*, have the running time $O(|\Sigma|\, n(\log m + |\Sigma|))$ and $O(\delta n \log m)$, respectively. Fig.1 shows that *δ-Version 2* is faster than *δ-Version 1* for a wide range of text size n. Nevertheless, the latter seems to come closer to the running time of the former in some cases. One such case is when $n = 4k$. Apart from few such places, *δ-Version 2* seems to outperform *δ-Version 1*. Furthermore, the difference in their running is very high, for example when $n = 6k$, *δ-Version 2* is 8.04 times faster than *δ-Version 1* (See Table I). The big difference is due to the encoding scheme used in *δ-Version 1* which expands the text and the pattern $|\Sigma|$ times longer.

This does not necessarily mean that we have to avoid using *δ-Version 1* all the times. The timing curves on Fig. 2(a) and Fig. 2(b) provide an evidence that *δ-Version 1* is the best choice when δ is very high or $|\Sigma|$ is very small. Furthermore, we could see that the alphabet used in practice is small.

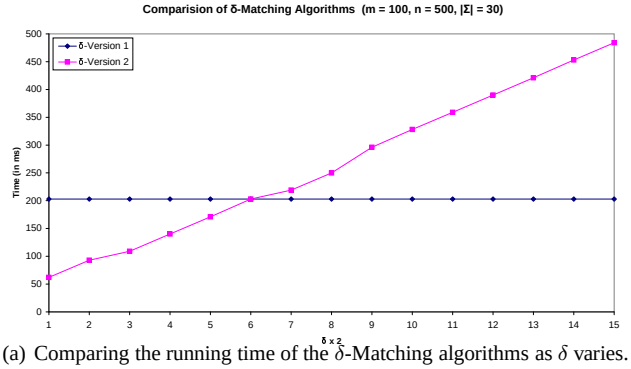
Thus, *δ-Version 2* is superior to *δ-Version 1* in general. However, *δ-Version 1* is a better choice when δ is large or $|\Sigma|$ is very small.

B. γ-Matching with Don't Cares

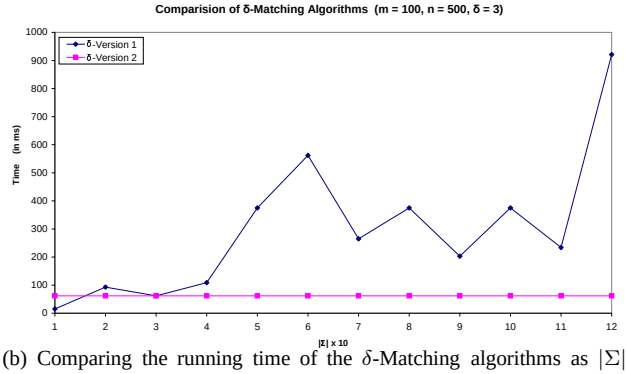
The two γ-Matching algorithms have running time $O(n\sqrt{m \log m})$ and $O(|\Sigma|\, n \log m)$ respectively. Fig. 1 shows that *γ-Version 1* always beats *γ-Version 2* in performance. Nonetheless, according to Table I, the ratio of their running time is almost constant, 0.29 on average. Besides, Fig. 2(c) illustrates the effect of $|\Sigma|$ in the running time. It clearly shows that when $|\Sigma|$ is below a certain value, 14 in this example, *γ-Version 2* is a better choice. Thus, *γ-Version 1* may be preferred to *γ-Version 2* for all cases, but when $|\Sigma|$ is small.

C. (δ,γ)-Matching with Don't Cares

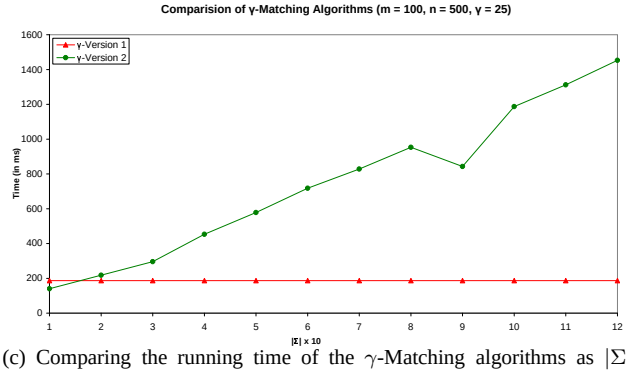
Fig. 1 and Table I show a surprising result about the running time of the *(δ,γ)-Version 1*. That is, solving



(a) Comparing the running time of the δ-Matching algorithms as δ varies.



(b) Comparing the running time of the δ-Matching algorithms as $|\Sigma|$ varies.



(c) Comparing the running time of the γ-Matching algorithms as $|\Sigma|$ varies.

Fig. 2. Timing curves for δ-, γ-Matching algorithms.

TABLE I

RUNNING TIME OF THE ALGORITHMS IN MILLISECONDS WHEN $m = 100, \delta = 3, \gamma = 25, |\Sigma| = 70$ AND *Number of "Don't Cares" in the Pattern* $= 5$

n	δ-Matching		γ-Matching		(δ,γ)-Matching
	Version 1	Version 2	Version 1	Version 2	Version 1
2k	1781	218	812	2938	375
4k	1015	796	2359	6359	1750
6k	4781	594	2218	7718	1093
8k	1891	735	2844	9703	1328
10k	2421	1250	3969	13641	1953
12k	2781	1171	4484	15297	2093
14k	11391	1453	5140	17610	2531
16k	4703	1515	5875	19641	2750
18k	5062	1719	6500	23000	3078
20k	6813	1906	7360	26000	3390
22k	5813	2000	7672	26546	3797
24k	6750	2281	8969	30610	4094

γ-Matching using the function technique, i.e. using an even periodic function and their cosine expansions for the computation, takes much longer than solving (δ, γ)-Matching using the same technique. But γ-Matching is just a part of (δ, γ)-Matching along with an instance of δ-Matching. The reason for this can be simply explained as follows: When the function technique is used, the supplementary γ-Matching in (δ, γ)-Matching takes $O(\delta nlogm)$ time, but the stand alone γ-Matching takes $O(|\Sigma|\, nlogm)$. Moreover, $|\Sigma|$ is much greater than δ in this example, and is usually the case.

Fig. 1 provides evidence of another interesting feature of (δ, γ)-*Version 1*. The vertical-difference between the curves corresponding to δ-*Version 2* and (δ, γ)-*Version 1* is very small. But (δ, γ)-*Version 1* executes δ-*Version 2* as its first step. This depicts that, only a small fraction of the time is spent in the supplementary γ-Matching. Having said this, it is obvious that no other combinations of δ- and γ-Matching can come even closer to (δ, γ)-*Version 1*.

VII. Conclusion

We have given experimental analysis of five now-best algorithms for approximate matching problems on string of integers allowing the presence of "don't cares". In particular, for a given text $T_{1..n}$ and a pattern $P_{1..m}$ with "don't cares", we have implemented two algorithms for computing the δ-Matching in $O(|\Sigma|n(\log m + |\Sigma|))$ and $O(\delta n \log m)$ time, respectively. Additionally, we have implemented two algorithm for computing the γ-Matching in $O(|\Sigma|nlogm)$ and $O(n\sqrt{m \log m})$ time respectively. Additionally, an $O(\delta n \log m)$-time algorithm for the (δ, γ)-Matching has also been implemented.

It remains as an open problem whether the techniques used in this paper can be implemented to solve other numeric string problems. One of such is the problem of *shift-matching*, where the values in the pattern can be shifted up or down by some constant when looking for a match.

References

[1] A. Amir, O. Lipsky and E. Porat. Approximate matching in the L_1 metric. *In Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, 2005.

[2] Y. J. Pinzon Ardila, M. Christodoulakis, C. S. Iliopoulos, and M. Mohamed. Efficient (δ, γ)-pattern-matching with don't cares. *In Proceedings of the 16th Australasian Workshop on Combinatorial Algorithms (AWOCA) (Ballarat, Australia)*, pages 27–38, 2005.

[3] M. J. Atallah. Faster Image Template Matching in the Sum of the Absolute Value of Differences Measure. *IEEE Transactions on Image Processing*, 10(4):659-663, 2001.

[4] BG92 R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[5] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics*, 79(11):1135–1148, 2002.

[6] P. Clifford, R. Clifford and C. S. Iliopoulos. Faster algorithms for δ, γ-matching and related problems. *In Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, 2005.

[7] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. *In 34th Symposium on Theory of Computing (STOC'02)*, pages 592–601, 2002.

[8] R. Cole, C. S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. On special families of morphisms related to δ-matching and don't care symbols. *Information Processing Letters*, pages 227-233, 2003.

[9] H. Cormen, C. E. Lieserson and R. L. Rivest. Introduction to Algorithms, *MIT Press*, 1990.

[10] M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. Pinzon. Approximate String Matching in Musical Sequences. *In Proceedings of the Prague Stringology Conference (PSC'01)*, pages 26–36, 2001.

[11] J. W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, 19:297301, 1965.

[12] M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the FFT. *Archive for History of Exact Sciences* 34(3):265-277, 1985.

[13] O. Lipsky. Efficient distance computations. Master's thesis, Bar-Ilan University, 2003.