Pattern Matching Algorithms with Don't Cares

Costas S. Iliopoulos and M. Sohel Rahman

Algorithm Design Group
Department of Computer Science, King's College London,
Strand, London WC2R 2LS, England
{csi,sohel}@dcs.kcl.ac.uk
http://www.dcs.kcl.ac.uk/adg

Abstract. In this paper, we present algorithms for pattern matching, where either the pattern $\mathcal{P}$ or the text $\mathcal{T}$ can contain "don't care" characters. If the pattern $\mathcal{P}$ contains don't care characters, then we can solve the pattern matching problem in $O(n + m + \alpha)$ time, where α is the total number of occurrences of the component subpatterns. We also can handle online queries, given an $O(n)$ preprocessing time, requiring $O(m + \alpha)$ time per query. If, on the other hand, the text $\mathcal{T}$ contains don't care characters, then we can solve the problem in $O(n + m + |occ(\mathcal{P})|)$ time where $|occ(\mathcal{P})|$ is the total number of occurrences of $\mathcal{P}$ in $\mathcal{T}$. The assumption that we make in this case is that the length of each component sub-text is greater than the length of the pattern.
Keywords: algorithm, pattern matching, don't care.

1 Introduction

The classical pattern matching problem is to find all the occurrences of a given pattern $\mathcal{P}$ of length m in a text $\mathcal{T}$ of length n, both being sequences of characters drawn from a finite character set Σ. This problem is interesting as a fundamental computer science problem and is a basic need of many applications, such as text retrieval, music retrieval, computational biology, data mining, network security, among many others. Several of these applications require, however, more sophisticated forms of searching. Pattern matching has been generalized to searching with error bounds, e.g. Hamming distance [3, 8, 15], edit distance [5, 15, 21]. These variations of the original problem are known as approximate pattern matching. In many, if not most, practical cases it is the approximate version of the pattern matching problem that turns out to be the most applicable one. Several other pattern matching problems have been considered within the approximate paradigm. Fischer and Paterson [7] generalized pattern matching to include don't cares: given a pattern $\mathcal{P}$ and a text $\mathcal{T}$, either of which may contain don't cares, denoted $*$, the goal is to output all occurrences of $\mathcal{P}$ in $\mathcal{T}$. The new dimension in the problem was the introduction of don't cares, also known as gaps in the literature, which matches any character in the alphabet i.e. $*$ matches every symbol $a \in \Sigma$. In [7] the don't care matching problem was solved in $O(n \log m \log \Sigma)$ time[1]. Since their (deterministic) algorithm in 1974, the only improvements, until recently, were by Muthukrishan and Palem [17], who reduced the constant factor. And in fact the string matching with don't cares problem is proved to be at least as hard as the boolean convolution problem [18]. Indyk [10], on the other hand gave a randomized algorithm that also involved convolutions, running in time $O(n \log n)$. Kalai [11] presented a slightly

[1] The running time reported in [7], $O(n \log^2 m \log \log m \log |\Sigma|)$, is slightly higher because they do not use the RAM model.

better but much simpler randomized algorithm which runs in $O(n \log m)$ time. Finally, Cole and Hariharan in [6], solved the long lasting open problem of removing the dependency on $|\Sigma|$ in the algorithm of [2] and presented a deterministic $O(n \log m)$ time algorithm. Their algorithm also used convolution.

Pinter [19] on the other hand avoided the use of convolution and used the Aho-Corasick algorithm [2] to solve the problem. The running time of Pinter's algorithm is $O(n + m + \alpha)$, where α is the total number of occurrences of the component subpatterns. Notably, however, Pinter's technique cannot be used to index the text. Very recently, Rahman et al. [20] presented an algorithm where they preprocess the text in optimal $O(n)$ time and can answer subsequent queries in $O(m + \alpha \log \log n)$ time. Cole et al., in [4], also presented an algorithm to solve this problem. The algorithm in [4] first preprocesses the text in $O(n \log^K n + n \log |\Sigma|)$ time to build a data structure of size $O(n \log^K n)$ which answers subsequent queries in time $O(2^K \log \log n + m + |occ(\mathcal{P})|)$. Here K is the number of don't cares in the pattern and $ooc(\mathcal{P})$ denotes the set of occurrences of the pattern in the text. As is evident from the running time, this algorithm is only effective when K is constant. All the results of [19], [20] and [4], mentioned above, assume only pattern to contain the don't care characters. To the best of our knowledge there hasn't been much progress in the literature to solve the problem having text (instead of pattern) containing don't cares without using the fast fourier transform.

In this paper we first present a new algorithm that efficiently finds a pattern with don't care characters in a text. We then present algorithm for the case when text has don't care characters instead of the pattern. The significance of our algorithms is twofold.

1. Firstly, almost all the algorithms to solve the don't care matching problem in the literature makes use of First Fourier Tranformation (FFT) technique which, unfortunately, doesn't perform well in practical cases due to its large hidden constant. Also, convolutions, along with other existing algorithms in the literature, are not conducive to indexing. Our algorithm does not make use of FFT technique and, as will be shown, they can index the text in linear time to handle online queries efficiently.
2. Secondly, there hasn't been much work without employing FFT technique in the literature on the version where the text has don't cares instead of the pattern. We here present an efficient algorithm (not using FFT) for this version as well. In this case, however, we make an assumption that the length of each component sub-text is greater than the length of the pattern.

The rest of the paper is organized as follows. In Section 2 we present the preliminary concepts. In Section 3 and 4 we present our new algorithms. Finally we conclude in Section 5.

2 Preliminaries

In what follows, we are considering a finite alphabet Σ. Assume that we are given a text $\mathcal{T} = \mathcal{T}[1] \ldots \mathcal{T}[n]$ of length n and a pattern $\mathcal{P} = \mathcal{P}[1] \ldots \mathcal{P}[m]$ of length m. We use the notation $\mathcal{T}[i..j], 1 \leq i \leq j \leq n$, to indicate the substring $\mathcal{T}[i]\ \mathcal{T}[i+1] \ldots\ \mathcal{T}[j]$ of $\mathcal{T}$. On the other hand by $\mathcal{T}^{-1}$ we denote the reverse string of $\mathcal{T}$ i.e. $\mathcal{T}^{-1} = \mathcal{T}[n] \ldots \mathcal{T}[1]$. The classical pattern matching problem consists in locating all the occurrences of $\mathcal{P}$ in $\mathcal{T}$, that is, all possible i such that for all j in $[1, m]$, $\mathcal{T}[i + j - 1] = \mathcal{P}[j]$. This problem has been extended in a very interesting way by introducing don't care characters as follows.

Definition 1. Don't Care Character. *A don't care character, denoted by '$*$' is a character such that $* \notin \Sigma$ and $*$ matches any character $a \in \Sigma$. So we say $* = a$ for all $a \in \Sigma$.*

Now let us define the problems we wish to tackle more formally.

Problem "DCP" (Don't Care in Pattern). *We are given a text $\mathcal{T}$ over the alphabet Σ and a pattern $\mathcal{P}$ over the alphabet $\Sigma \cup \{*\}$. The problem is to find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$.*

Problem "DCT" (Don't Care in Text). *We are given text $\mathcal{T}$ over the alphabet $\Sigma \cup \{*\}$ and a pattern $\mathcal{P}$ over the alphabet Σ. The problem is to find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$.*

Example 1. Suppose we have a text $\mathcal{T}$ and a pattern $\mathcal{P}$:
$\mathcal{T} = A\ C\ C\ G\ G\ A\ A\ G\ G\ T\ A\ A\ G\ T\ C\ G\ T\ A\ A\ A\ T\ T$
$\mathcal{P} = C\ G\ *\ A\ A\ *\ *\ T$

Note that $\mathcal{P}[3]$ can match any character in the $\mathcal{T}$ as can $\mathcal{P}[6]$ and $\mathcal{P}[7]$. It can easily be verified that in this case we have two occurrences of $\mathcal{P}$ in $\mathcal{T}$, starting at positions 3 and 15.

In our discussion it will be useful to define a pattern $\mathcal{P}$ having don't care characters as follows.

Definition 2. Pattern with don't care characters. *A pattern $\mathcal{P}$ with don't care characters consists of subpatterns $\mathcal{P}_i,\ 1 \leq i \leq \ell$, where each subpattern $\mathcal{P}_i$ is a string over the alphabet Σ. For each $1 \leq i \leq \ell - 1$, we have a parameter k_i which indicates the number of don't care characters between $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$.*

Therefore, we can view a pattern $\mathcal{P}$ as follows:

$$\mathcal{P} = \mathcal{P}_1 *^{k_1} \mathcal{P}_2 *^{k_2} ... *^{k_{\ell-1}} \mathcal{P}_\ell$$

We denote by $occ(\mathcal{P}_i)$ the set of indices where $\mathcal{P}_i$ occurs in $\mathcal{T}$. So, $occ(\mathcal{P}_i) = \{r \mid \mathcal{P}_i = \mathcal{T}[r..r+|\mathcal{P}_i|-1]\}$. We define α to be the total number of occurrences of $\mathcal{P}_i, 1 \leq i \leq \ell$ in $\mathcal{T}$, i.e. $\alpha = \sum_{1 \leq i \leq \ell} |occ(\mathcal{P}_i)|$, where as by β we denote the number of occurrences of $\mathcal{P}$ in $\mathcal{T}$, i.e. $\beta = |occ(\mathcal{P})|$. The following facts are easily observable.

Fact 1. $\alpha = \Omega(\beta)$

Fact 2. $\beta = O(n)$

Fact 3. $\alpha = O(ln)$

Similar to the definition of $\mathcal{P}$ (Definition 2) we define a text $\mathcal{T}$ having don't care as follows.

Definition 3. Text with don't care characters. *A text $\mathcal{T}$ with don't care characters consists of subtexts $\mathcal{T}_i,\ 1 \leq i \leq \ell$, where each subtext T_i is a string over the alphabet Σ. For each $1 \leq i \leq \ell - 1$ we have a parameter k_i which indicates the number of don't care characters between $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$.*

In both the cases the total number of don't care characters is $K = \sum_{1 \leq i \leq \ell-1} k_i$. For the pattern having don't care characters, we define $m = \sum_{1 \leq i \leq \ell-1} |\mathcal{P}_i|$ and $m' = |\mathcal{P}| = m + K$. Similarly, for the text with don't care characters, we define $n = \sum_{1 \leq i \leq \ell-1} |\mathcal{T}_i|$ and $n' = |\mathcal{T}| = n + K$. In this paper, we present algorithms to solve both Problem DCP and DCT.

3 Don't Care in Pattern

In this section we first present an algorithm to solve Problem DCP. We basically use a counting argument as follows. We preprocess the occurrences of the subpatterns $\mathcal{P}_i, 1 \leq i \leq \ell$ and construct an array *permitted* of length n such that $permitted[i] = \ell$, if and only if, for all $1 \leq j \leq \ell$ there exists an $r_j \in occ(\mathcal{P}_j)$ such that $r_j - \sum_{1 \leq q < j}(|\mathcal{P}_q| + |k_q|) = i$. It is easy to see that with the array *permitted* duly filled we can set $occ(\mathcal{P}) = \{i \mid permitted[i] = \ell\}$. The algorithm is formally given in the form of Algorithm 1.

Algorithm 1 Algorithm to Solve Problem DCP

1: Build a Suffix Array SA for $\mathcal{T}$
2: **for** $i = 1$ to n **do**
3: permitted[i] = 0 {initializing the *permitted* array}
4: **end for**
5: Set $Val[1] = 0$
6: **for** $i = 2$ to ℓ **do**
7: $Val[i] = Val[i-1] + |\mathcal{P}_{i-1}| + k_{i-1}$
8: **end for**
9: $occ(\mathcal{P}) = \emptyset$
10: **for** each $\mathcal{P}_i, 1 \leq i \leq \ell$ **do**
11: Compute $occ(\mathcal{P}_i)$ using SA.
12: **for** each $r \in occ(\mathcal{P}_i)$ **do**
13: $permitted[r - Val[i]] = permitted[r - Val[i]] + 1$ {Assume that boundary conditions are checked}
14: **if** $permitted[r - Val[i]] = \ell$ **then**
15: $occ(\mathcal{P}) = occ(\mathcal{P}) \bigcup (r - Val[i])$
16: **end if**
17: **end for**
18: **end for**
19: **return** $occ(\mathcal{P})$

Let us now analyze Algorithm 1. Step 1 can be done in $O(n)$ using one of the very recent linear algorithms of [12–14]. Step 2 initializes the *permitted* array and requires $O(n)$ time. In the 'FOR' loop of Step 10, we first compute $occ(\mathcal{P}_i)$ for all $1 \leq i \leq \ell$ (Step 11) and then for each $r \in occ(\mathcal{P}_i)$, we do constant time manipulation in the *permitted* array. The constant time manipulation is warranted by the construction of the array Val in Step 6 requiring $O(\ell)$ time. Computing all $occ(\mathcal{P}_i)$ and the $O(1)$ time manipulation per each element of all $occ(\mathcal{P}_i)$ can be performed in $O(\sum_{1 \leq i \leq \ell} |\mathcal{P}_i| + \sum_{1 \leq i \leq \ell} |occ(\mathcal{P}_i)|) = O(m + \alpha)$ time using the algorithms of [1]. So the total running time of our algorithm would be $O(n + m + \alpha)$.

Theorem 4. *Algorithm 1 solves Problem DCP in $O(n + m + \alpha)$ time.*

3.1 Indexing

The indexing problem is a very important variant of the pattern matching problem where the given text has to be preprocessed so that batch of queries can be answered efficiently. The problem is formally defined as follows:

Problem "IDCP" (Indexing with Don't Care in Pattern). *We are given text $\mathcal{T}$ over the alphabet Σ that can be preprocessed to answer the following form of queries:*
Query: Given a pattern $\mathcal{P}$ over the alphabet $\Sigma \cup \{*\}$, find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$.

We can adapt the idea used in Algorithm 1 to solve Problem IDCP. The direct use of Algorithm 1 to answer the queries would require us to use Step 2 to 19 to answer each query requiring $O(n+m+\alpha)$ time. This would mean, even if $\alpha = o(n)$, we still have $O(n+m)$ time per query. We would prefer a query time of $O(m+\alpha)$ to avoid direct dependence on n. Note that, we have the n term in the query running time due to the initialization of the array *permitted* in Step 2. In order to avoid this $O(n)$ initialization step we observe the following key fact:

Fact 5. $occ(\mathcal{P}) \subseteq occ(\mathcal{P}_i), 1 \leq i \leq l$.

Corollary 1. $occ(\mathcal{P}) \subseteq occ(\mathcal{P}_1)$.

Hence, we can ignore all $permitted[i]$ such that $i \notin occ(\mathcal{P}_1)$. We avoid initializing the whole *permitted* array as follows. As part of the index structure we first initialize the *permitted* array to ensure that $permitted[i] = 0, 1 \leq i \leq n$. So the index construction consists of Step 1 to 4 of Algorithm 1. During a query we only use those indices i of *permitted* such that $i \in occ(\mathcal{P}_1)$ and when we are done we (re)initialize those indices for the next round. In this way, instead of $O(n)$ work, for the initialization of *permitted*, we only spend $O(|occ(\mathcal{P}_1)|)$ time per query. The steps are formally stated in Algorithm 2.

We can see that the index consists of an suffix array of the text and the initialized array *permitted* requiring, in total, $O(n)$ time for construction. The query time is $O(m+\alpha)$ as follows. Step 2 requires $O(\ell) = O(m)$ time. Step 6 and 7 can be performed in $O(|\mathcal{P}_1| + |occ(\mathcal{P}_1)|)$ time. The whole for loop of Step 10 requires $O((m - |\mathcal{P}_1|) + (\alpha - |occ(\mathcal{P}_1)|))$ time. Finally, Step 21 requires $O(|occ(\mathcal{P}_1)|)$ time. So, in total, the query time is $O(m+\alpha)$ as required.

Theorem 6. *Given a text $\mathcal{T}$ over the alphabet Σ, we can preprocess $\mathcal{T}$ in optimal $O(n)$ time and space so that we can report the occurrences of a pattern $\mathcal{P}$ over alphabet $\Sigma \cup \{*\}$ in $O(m+\alpha)$ time.*

3.2 Comparison

The algorithm we have presented (Algorithm 1) has the same running time as Pinter's algorithm [19] and, in fact, suffers from the same drawback as the one in [19]. As is evident from Fact 3, in the worst case the running time can be $O(n+m+nl)$. However, unlike Pinter's algorithm, our algorithm (Algorithm 2) gives us the ability to process patterns in an online fashion, one after another. Given a preprocessing time of $O(n)$ for the text $\mathcal{T}$ (Step 1 to 4 of Algorithm 1), the occurrences of a pattern can be reported in $O(m+\alpha)$ using our technique (Algorithm 2).

As is already stated in Section 1, Cole et al. provided a solution for Problem DCP in [4]. The algorithm in [4] first preprocesses the text $\mathcal{T}$ in $O(n \log^K n + n \log |\Sigma|)$ time to build a data structure of size $O(n \log^K n)$ which answers subsequent queries in time $O(2^K \log\log n + m + \beta)$. As is evident from the running time, this algorithm is only effective when K is constant. On the other hand, our algorithm requires only $O(n)$ time and space to preprocess to build a space-efficient suffix array along with a simple array (*permitted*). Also both our preprocessing and query time are independent of K. Our query time is $O(m+\alpha)$ as opposed to $O(2^K \log\log n + m + |occ(\mathcal{P})|)$ of [4]. The only problem that we have is the presence of the parameter α instead of $|occ(\mathcal{P})|$ in our query time.

Algorithm 2 Query Algorithm

```
1: Set Val[1] = 0
2: for i = 2 to ℓ do
3:    Val[i] = Val[i − 1] + |P_{i−1}| + k_{i−1}
4: end for
5: occ(P) = ∅
6: Compute occ(P_1) using SA.
7: for each r ∈ occ(P_1) do
8:    permitted[r] = 1
9: end for
10: for each P_i, 2 ≤ i ≤ ℓ do
11:    Compute occ(P_i) using SA.
12:    for each r ∈ occ(P_i) do
13:       if permitted[r − Val[i]] ≥ 1 {Checking whether (r − Val[i]) ∈ occ(P_1)} then
14:          permitted[r − Val[i]] = permitted[r − Val[i]] + 1
15:          if permitted[r − Val[i]] = ℓ then
16:             occ(P) = occ(P) ⋃ (r − Val[i])
17:          end if
18:       end if
19:    end for
20: end for
21: for each r ∈ occ(P_1) do
22:    permitted[r] = 0 {Re-initializing permitted array}
23: end for
24: return occ(P)
```

We believe, however, that in many cases our query time will outperform theirs because their query time is exponential in the number of don't cares, K. And, in fact, in many practical applications, including biological instances, K can be $O(m)$. Finally our algorithm is much more simpler than that of [4]. One final remark is that we can use suffix tree instead of suffix array in the preprocessing step virtually without any modification at all in Algorithm 1. The reason we preferred suffix array over suffix tree is because the former is more space-efficient than the latter.

4 Don't Care Characters in Text

In this section we present a solution for Problem DCT. We however make an assumption that $|\mathcal{T}_i| > m, 1 \leq i \leq \ell$. We first define some notations that we use to describe our algorithm. We denote by $occ_{\mathcal{T}}(\mathcal{P})$ the set of occurrences of $\mathcal{P}$ in $\mathcal{T}$. In our algorithm we first construct a generalized suffix tree [9] of $\mathcal{T}_i, 1 \leq i \leq \ell$ and construct the sets $occ_{\mathcal{T}_i}(\mathcal{P}), 1 \leq i \leq \ell$. We then update the indices in sets $occ_{\mathcal{T}_i}(\mathcal{P}), 2 \leq i \leq \ell$ with respect to $\mathcal{T}$. It is easy to see that after the update we have $\bigcup_{1 \leq i \leq \ell} occ_{\mathcal{T}_i}(\mathcal{P}) \subseteq occ(\mathcal{P})$. Hence we now have to find out the set $V = occ(\mathcal{P}) \setminus \bigcup_{1 \leq i \leq \ell} occ_{\mathcal{T}_i}(\mathcal{P})$. It is easy to see that each $j \in V$ comes from 3 cases.

Case 1. If there occurs a prefix $\mathcal{P}[1..r]$ ending at the last position of $\mathcal{T}_i$ and $k_i \geq m - r$.

Case 2. If there occurs a prefix $\mathcal{P}[1..r_1]$ ending at the last position of $\mathcal{T}_i$, a suffix $\mathcal{P}[m - r_2 + 1..m]$ starting at the first position of $\mathcal{T}_{i+1}$ and $k_i = m - r_1 - r_2$.

Case 3. If there occurs a suffix $\mathcal{P}[m - r + 1..m]$ starting at the first position of $\mathcal{T}_i$ and $k_{i-1} \geq m - r$.

We compute the set V as follows. Let us define the sets Pre_j and Suf_j such that $Pre_j = \{r \mid \mathcal{P}[1..r] \text{ occurs at position } j-r+1\}$ and $Suf_j = \{r \mid \mathcal{P}[m-r+1..m] \text{ occurs at position } j\}$. The idea is to first construct $Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$ and $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$ for each pair of $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$ and then check for the occurrences of the above-mentioned three cases.

In what follows we describe the steps with respect to one pair, $\mathcal{T}_i, \mathcal{T}_{i+1}$. The natural idea would be to construct Aho-Corasic automata [2] AC_{suf} and AC_{pre}, respectively, for all the suffixes and prefixes of $\mathcal{P}$. Using the Aho-Corasic Pattern Matching Machine (PMM) [2] we scan each pair of $\mathcal{T}_i$ and $\mathcal{T}_{i+1}, 1 \leq i < \ell$ as follows. We use AC_{pre} for $\mathcal{T}_i$ and find all the prefixes of $\mathcal{P}$ ending at the last position of $\mathcal{T}_i$ and construct the set $Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$. On the other hand we use AC_{suf} for $\mathcal{T}_{i+1}$ and find all the suffixes of $\mathcal{P}$ starting at the first position of $\mathcal{T}_{i+1}$ and construct the set $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$.

This idea would work but the problem is that the constructions of the two automata would require $O(m^2)$. The essential thing to note here is that we are interested in only 2 particular positions, namely, the last position of $\mathcal{T}_i$ and the first position of $\mathcal{T}_{i+1}$. As a result the use of PMM with the two automata is too general a solution in our context spending, perhaps, more computational effort than required. Indeed, as we shall see, we can perform our required task in $O(m)$ time as follows. We first describe how we can construct the set $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$. In what follows we assume that the reader is familiar with suffix tree [16, 22]. We build a suffix tree $ST_{\mathcal{P}}$ of $\mathcal{P}[2..m]$. We start traversing $ST_{\mathcal{P}}$ as we scan $\mathcal{T}_{i+1}$. During this traversal, if at some node, we get a \$ transition this means we have found a suffix and we put the length of the suffix in $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$ and continue to traverse. As soon as we get stuck at a node in $ST_{\mathcal{P}}$ we stop. we can get stuck in two ways: 1. We can reach the end of the tree which off course means that we found a suffix as well. 2. We reach a node where there are no outgoing transitions matching the character we are currently scanning in $\mathcal{T}_{i+1}$. It is easy to see that this procedure would give us the set $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$ in $O(m)$ time. This follows from the facts that construction of $ST_{\mathcal{P}}$ requires $O(m)$ and at most $m-1$ characters from $\mathcal{T}$ need be scanned.

To construct $Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$, recall that we want to find out the prefixes of $\mathcal{P}$ ending at $\mathcal{T}_i[|\mathcal{T}_i|]$. What we do is follows. We build a suffix tree $ST_{\mathcal{P}^{-1}}$ of $\mathcal{P}^{-1}[2..m]$ and repeat the above procedure on $\mathcal{T}_i^{-1}$. To see that this will work we just need to realize that we want to find out the suffixes of $\mathcal{P}^{-1}[2..m]$ starting at $\mathcal{T}_i^{-1}[1]$.

To cover Case 1, we check whether $k_i \geq m-r$ for each r in $Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$. If yes, it is easy to see that we have an occurrence of $\mathcal{P}$ at position $\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i-r+1$ of $\mathcal{T}$. To cover Case 3, we check whether $k_{i-1} \geq m-r$ for each r in $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$. If yes, it is easy to see that we have an occurrence of $\mathcal{P}$ at position $\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1-m+r$ of $\mathcal{T}$. Finally to cover case 2 we check whether there is an r_1 in $Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$ and r_2 in $Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$ such that $k_i = m-r_1-r_2$. If yes then it is easy to see that we have an occurrence of $\mathcal{P}$ at position $\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i-r_1+1$ of $\mathcal{T}$. In this case we say that $r_1 \in Pre_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)-a_i}$ has a *matching entry* $r_2 \in Suf_{\sum_{1\leq j\leq i}(|\mathcal{T}_j|+a_j)+1}$. Now that we have found the set V we are done. The steps of the algorithm is formally given in Algorithm 3.

Let us now analyze the running time of Algorithm 3. Step 1 can be done in $O(\sum_{1\leq i\leq \ell}|\mathcal{T}_i|) = O(n)$ time. In Step 7 we first find out the occurrences of $\mathcal{P}$ in the strings $\mathcal{T}_i, 1 \leq i \leq \ell$ using GST and then update the corresponding indices of occurrences with respect to $\mathcal{T}$. Using the array Val, constructed in Step 4 in $O(\ell)$ time, this can be done in $O(m+|occ(\mathcal{P})|)$ because $\sum_{1\leq i\leq \ell}|occ_{\mathcal{T}_i}(\mathcal{P}) \leq |occ(\mathcal{P})|$. Note that up to this point we have found out all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ excluding those involving

Algorithm 3 Algorithm to Solve Problem DCT

1: Build a generalized suffix tree GST for the strings $\mathcal{T}_i, 1 \leq i \leq \ell$.
2: Set $occ(P) = \emptyset$
3: Set $Val[1] = 0$
4: **for** $i = 2$ to ℓ **do**
5: $\quad Val[i] = Val[i-1] + |\mathcal{T}_{i-1}| + k_{i-1}$
6: **end for**
7: **for** $1 \leq i \leq \ell$ **do**
8: $\quad$ Construct $occ_{\mathcal{T}_i}(\mathcal{P})$.
9: $\quad$ **for** each $j \in occ_{\mathcal{T}_i}(\mathcal{P})$ **do**
10: $\quad\quad$ Set $j = j + Val[i-1]$
11: $\quad\quad$ $occ(P) = occ(P) \bigcup j$
12: $\quad$ **end for**
13: **end for**
14: Construct a suffix tree $ST_{\mathcal{P}}$ of $\mathcal{P}[2..m]$
15: Construct a suffix tree $ST_{\mathcal{P}^{-1}}$ of $\mathcal{P}^{-1}[2..m]$
16: **for** $1 \leq i \leq \ell$ {Assume that boundary conditions are checked} **do**
17: $\quad$ Construct the set $Pre_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)-a_i}$.
18: $\quad$ Construct the set $Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}$.
19: **end for**
20: **for** $1 \leq i \leq \ell$ {Assume that boundary conditions are checked} **do**
21: $\quad$ **for** each r in $Pre_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)-a_i}$ **do**
22: $\quad\quad$ **if** $k_i \geq m - r$ **then**
23: $\quad\quad\quad$ $occ(P) = occ(P) \bigcup \sum_{1 \leq j \leq i}(|\mathcal{T}_j| + a_j) - a_i - r + 1$
24: $\quad\quad$ **else**
25: $\quad\quad\quad$ **if** there is a matching entry in $Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}$ **then**
26: $\quad\quad\quad\quad$ $occ(P) = occ(P) \bigcup \sum_{1 \leq j \leq i}(|\mathcal{T}_j| + a_j) - a_i - r + 1$.
27: $\quad\quad\quad$ **end if**
28: $\quad\quad$ **end if**
29: $\quad$ **end for**
30: $\quad$ **for** each r in $Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}$ **do**
31: $\quad\quad$ **if** $k_{i-1} \geq m - r$ **then**
32: $\quad\quad\quad$ $occ(P) = occ(P) \bigcup \sum_{1 \leq j \leq i}(|\mathcal{T}_j| + a_j) - r + 1$.
33: $\quad\quad$ **end if**
34: $\quad$ **end for**
35: **end for**
36: **return** $occ(\mathcal{P})$

the don't cares in $\mathcal{T}$. Step 14 and Step 15 each require $O(m)$ time. In the 'For' loop of Step 16 we consider each pair of $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$. For each such pair we perform Step 17 and Step 18 which takes $O(m)$ as is deduced before. So the total time required for Step 16 is $O(lm)$. Notably, $lm < n$ due to our assumption that $|t_i| > m, 1 \leq i \leq \ell$. Finally in Step 20 we find out the set V i.e all the occurrences due to Case 1, 2 and 3. It is clear that $|Pre_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)-a_i}| < m$ and $|Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}| < m$ as well. So for a particular pair we can check Case 1 (Step 22) and Case 3 (Step 25) in $O(m)$. But checking Case 3 (Step 25) is not that trivial. Nevertheless, we can do it in $O(m)$ as follows. We can implement the set $|Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}|$ using an array A of length $(m-1)$ where $A[j] = 1$ means the occurrence of $\mathcal{P}[m-j+1..m]$. So whether there is a matching entry in $|Suf_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)+1}|$ for

a $r \in |Pre_{\sum_{1 \leq j \leq i}(|\mathcal{T}_j|+a_j)-a_i}|$ can be found in constant time. So the total time required for Step 20 can be done in $O(lm)$. So we get the following theorems.

Theorem 7. *Algorithm 3 solves Problem DCT in $O(n + m + |occ(\mathcal{P})|)$ time.*

5 Conclusion

In this paper we have presented algorithms for pattern matching where either the pattern $\mathcal{P}$ or the text $\mathcal{T}$ can contain don't care characters. If the pattern $\mathcal{P}$ contains don't care characters then we can report all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ in $O(n + m + \alpha)$ time. Also, our algorithm can deal with batch of queries (in $O(m + \alpha)$ time per query) once the linear preprocessing is done. If, on the other hand, the text $\mathcal{T}$ contains don't care characters, then we can solve the problem in $O(n + m + |occ(\mathcal{P})|)$ time. The assumption that we make in this case is that the length of each component sub-text is greater than the length of the pattern. A number of issues remain as possible candidates for future research as follows:

1. It is clear that the bottleneck in the running time of Algorithm 1 is the presence of the term α. It is desirable to replace α with an $o(n)$ term which, however, doesn't seem to be very easy to accomplish without paying much more cost in either the preprocessing time or the query time itself or in both of them. An example of this situation can be seen in the running time of Cole et. al [4] (see discussion in Section 3.2). We are currently exploring some heuristic techniques to apply on Algorithm 1. With these modifications, we strongly believe that, our algorithm would outperform all the existing algorithms for this problem in practice.
2. For Algorithm 3, obviously, the next step should be to try to lift our assumption that the length of each component sub-text is greater than the length of the pattern.

References

1. Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In *SPIRE*, pages 31–43, 2002.
2. A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
3. Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with mismatches. In *SODA*, pages 794–803, 2000.
4. Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.
5. Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. In *SODA*, pages 463–472, 1998.
6. Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC*, pages 592–601, 2002.
7. M.J. Fischer and M.S. Paterson. String matching and other products. *in Complexity of Computation, R.M. Karp (editor), SIAM AMS Proceedings*, 7:113–125, 1974.
8. Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.
9. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

10. P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. *Proceedings of the 39th Symposium on Foundations of Computer Science*, 1998.
11. A. Kalai. Efficient pattern-matching with don't cares. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 655–656, 2002.
12. Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *ICALP*, pages 943–955, 2003.
13. Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *CPM*, pages 186–199, 2003.
14. Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *CPM*, pages 200–210, 2003.
15. Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.
16. Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
17. S. Muthukrishan and K. Palem. Non-standard stringology: Algorithms and complexity. *Proceedings of the 26th Symposium on the Theory of Computing, Canada*, 1994.
18. S. Muthukrishan and H. Ramesh. Non-standard stringology: Algorithms and complexity. *Information and Computation*, 122(1):140–148, 1995.
19. R.Y. Pinter. Efficient string matching with dont care patterns. *In A. Apostolico and Z. Galil (Eds.). Combinatorial algorithms on words, NATO Advanced Science Institute Series F: Computer and System Sciences*, 12:11–29, 1985.
20. M. Sohel Rahman, Costas Iliopoulos, Inbok Lee, Manal Mohamed, and W.F. Smyth. Finding patterns with variable length gaps or don't cares. In D.Z. Chen and D.T. Lee, editors, *COCOON*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006.
21. S.C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *FOCS*, pages 320–328, 1996.
22. Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.