Pattern matching with don't cares and few errors

Raphaël Clifford
University of Bristol, Dept. of Computer Science
Bristol, BS8 1UB, UK
clifford@cs.bris.ac.uk

Klim Efremenko
Bar-Ilan University, Dept. of Computer Science
52900 Ramat-Gan and
Weizmann Institute
Rehovot 76100, Israel
klimefrem@gmail.com

Ely Porat*
Bar-Ilan University, Dept. of Computer Science
52900 Ramat-Gan, Israel
porately@cs.biu.ac.il

Amir Rothschild
Bar-Ilan University, Dept. of Computer Science
52900 Ramat-Gan, Israel
amirrot@gmail.com

Abstract

We present solutions for the k-mismatch pattern matching problem with don't cares. Given a text t of length n and a pattern p of length m with don't care symbols and a bound k, our algorithms find all the places that the pattern matches the text with at most k mismatches. We first give an $\Theta\left(n(k + \log m \log k) \log n\right)$ time randomised algorithm which finds the correct answer with high probability. We then present a new deterministic $\Theta\left(nk^2 \log^2 m\right)$ time solution that uses tools originally developed for group testing. Taking our derandomisation approach further we develop an approach based on k-selectors that runs in $\Theta\left(nk \,\mathrm{polylog} m\right)$ time. Further, in each case the location of the mismatches at each alignment is also given at no extra cost.

*Research supported in part by the Binational Science Foundation (BSF)

Dagstuhl Seminar Proceedings 09281
Search Methodologies
http://drops.dagstuhl.de/opus/volltexte/2009/2244

1 Introduction

In this paper we consider string matching under the widely used Hamming distance and in particular a bounded version of this problem which we call *k-mismatch with don't cares*. Given a text t of length n and a pattern p of length m both possibly containing single character promiscuously matching or *don't care symbols* and a bound k, the problem is to find all the places that the pattern matches the text with at most k mismatches. If the distance is greater than k, the algorithm need only report that fact and not give the actual Hamming distance.

The problem of finding all the occurrences of a given pattern of length m in a text t of length n is a classic problem in computer science and can be solved in $\Theta(n)$ time [BM77, KMP77]. The problem of determining the time complexity of exact matching with optional single character *don't care* symbols has also been well studied. Fischer and Paterson [FP74] presented the first solution based on fast Fourier transforms (FFT) with an $\Theta(n \log m \log |\Sigma|)$ time algorithm in 1974[1], where Σ is the alphabet that the symbols are chosen from. Subsequently, the major challenge has been to remove this dependency on the alphabet size. Indyk [Ind98] gave a randomised $\Theta(n \log n)$ time algorithm which was followed by a simpler and slightly faster $\Theta(n \log m)$ time randomised solution by Kalai [Kal02]. In 2002, the first deterministic $\Theta(n \log m)$ time solution was given [CH02] which was then further simplified in [CC07].

The key observation given by [CC07] but implicit in previous work is that for numeric strings, if there are no don't care symbols then for each alignment of the pattern with respect to the text $1 \leq i \leq n - m + 1$ we can calculate

$$\sum_{j=1}^{m}(p_j - t_{i+j-1})^2 = \sum_{j=1}^{m}(p_j^2 - 2p_j t_{i+j-1} + t_{i+j-1}^2) \tag{1}$$

in $\Theta(n \log m)$ time using FFTs. Wherever there is an exact match this sum will be exactly 0. If p and t are not numeric, then an arbitrary one-to-one mapping can be chosen from the alphabet to the set of positive integers $\mathbb{N}$. In the case of matching with don't cares, each don't care symbol in p or t is replaced by a 0 and the sum is modified to be

$$\sum_{j=1}^{m} p'_j t'_{i+j-1}(p_j - t_{i+j-1})^2$$

where $p'_j = 0$ ($t'_i = 0$) if p_j (t_i) is a don't care symbol and 1 otherwise. This sum equals 0 if and only if there is an exact match with don't cares and can

[1]Throughout this paper we assume the RAM model with multiplication when giving the time complexity of the FFT. This is in order to be consistent with the large body of previous work on pattern matching with FFTs.

also be computed in $\Theta(n \log m)$ time using FFTs.

1.1 Related work and previous results

Much progress has been made in finding fast algorithms for the k-mismatch problem *without* don't cares over the last 20 years. $\Theta\left(n\sqrt{m \log m}\right)$ time solutions for the k-mismatch problem based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [Abr87, Kos87]. Their algorithms are in fact independent of the bound k and report the Hamming distance at every position irrespective of its value. In 1985 Landau and Vishkin gave a beautiful $\Theta(nk)$ algorithm that is not FFT based which uses constant time lowest common ancestor (LCA) operations on the suffix tree of p and t [LV86]. This was subsequently improved in [ALP04] to $\Theta\left(n\sqrt{k \log k}\right)$ time by a method based on filtering and FFTs again. Approximations within a multiplicative factor of $(1 + \epsilon)$ to the Hamming distance can also be found in $O(n/\epsilon^2 \log^c m)$ time, where the exact value of c depends on whether the algorithm is deterministic or randomised [Kar93] .

For a more limited version of the k-mismatch problem with don't cares, where don't cares are only permitted in either the pattern or text, but not both, a filtering algorithm has been developed which runs in $O(nm^{1/3}k^{1/3} \log^{2/3} m)$ time [CP07]. A variant of the edit-distance problem (see e.g. [LV85]) called the k-difference problem with don't cares was considered in [Aku95]. Progress has also been made recently on the related problem of indexing with errors and don't cares [CGL04, CLS$^+$06].

To the authors' knowledge, no previous efficient algorithms have been given to date for the k-mismatch problem with don't cares. However, the $\Theta\left(n\sqrt{m \log m}\right)$ divide and conquer algorithm of Kosaraju and Abrahamson can be easily extended to handle don't cares in both the pattern and text with little extra work. This is because the algorithm counts matches and not mismatches. First we count the number of non don't care matches at each position i in $\Theta\left(n\sqrt{m \log m}\right)$ time. Then we need only subtract this number from the maximum possible number of non don't care matches in order to count the mismatches. To do this we create a new pattern string p' so that $p'_j = 1$ if p_j is not a don't care and $p_j = 0$ otherwise. A new text string t' is also made in the same way. The cross-correlation of p' and t' now gives us the maximum number of non don't care matches possible at each position. This single cross-correlation calculation takes $\Theta(n \log m)$ time. Therefore the overall running time remains $\Theta\left(n\sqrt{m \log m}\right)$.

2 Our results

We present fast randomised solutions for the k-mismatch problem with don't cares. This problem does not appear to be amenable to any of the recent methods for solving the corresponding problem without don't cares. For example, the LCA based technique of Landau and Vishkin [LV86] requires the use of suffix trees to find longest common prefix matches between strings in constant time. It is not known how to find longest common prefixes in even sublinear time when arbitrary numbers of don't cares are allowed. Similarly, it does not appear to be possible to apply filtering methods such as those in [ALP04] when don't cares are permitted in both the pattern and the text.

We give two new algorithms that overcome these obstacles and provide substantial improvements to the known time complexities of the problem. We present a randomised algorithm in Section 4 that runs in $\Theta\left(n(k + \log m \log k) \log n\right)$ time and gives the correct answer with high probability. The basic technique is to repeatedly sample subpatterns of p and find the positions of single mismatches in the text. A subpattern is simply a copy of the pattern with some positions set to the don't care character. In order to count the total number of mismatches overall we are also required at each stage to find the position and values of all the mismatches found so far.

We then give a deterministic algorithm in Section 5 that runs in $\Theta\left(nk^2 \log^2 m\right)$ time. This algorithm uses a recent group testing result in a novel way, not as a stand alone tool, but rather as a derandomisation scheme. The testing scheme allows us to effectively choose subpatterns deterministically which in turn give us the required single mismatches. Combining the results of these tests enables us to find the locations of the mismatches and also to check that none has been missed out. Finally we discuss the use of k-selectors instead of group testing and show that a deterministic $\Theta\left(nk \operatorname{polylog} m\right)$ time solution to the k-mismatch problem with don't cares can be found.

3 Preliminaries

Let Σ be a set of characters which we term the *alphabet*, and let ϕ be the don't care symbol. Let $t = t_1 t_2 \dots t_n \in \Sigma^n$ be the text and $p = p_1 p_2 \dots p_m \in \Sigma^m$ the pattern. Both the pattern and text may also include ϕ in their alphabet depending on the problem definition. The terms *symbol* and *character* are used interchangeably throughout. Similarly, we will sometimes refer to a *location* in a string and synonymously at other times a *position*.

- Define $HD(i)$ to be the Hamming distance between p and $t[i, \dots, i+m-1]$ and define the don't care symbol to match any symbol in the alphabet.

- Define $HD_k(i) = \begin{cases} HD(i) & \text{if } HD(i) \leq k \\ \perp & \text{otherwise} \end{cases}$
- We say that there is a k-mismatch between p and t at alignment i if $HD_k(i) \neq \perp$.

Our algorithms make extensive use of the fast Fourier transform (FFT). An important property of the FFT is that in the RAM model, the cross-correlation,

$$(t \otimes p)[i] \stackrel{\text{def}}{=} \sum_{j=1}^{m} p_j t_{i+j-1}, \quad 0 \leq i \leq n-m+1,$$

can be calculated accurately and efficiently in $\Theta(n \log n)$ time (see e.g. [CLR90], Chapter 32). By a standard trick of splitting the text into overlapping substrings of length $2m$, the running time can be further reduced to $\Theta(n \log m)$. We will often assume that the text is of length $2m$ in the presentation of an algorithm or analysis and that the reader is familiar with this splitting technique.

With high probability

The main results presented in this paper are for randomised algorithms which give the correct answer *with high probability*. This term is used with varying meanings in the literature and our use provides particularly strict bounds. The motivation for our definition lies in the assumption that any pattern matching algorithm may be applied a large (typically polynomial) number of times over varying data sets. In this situation, it is desirable that the bounds on the probability of failure will still hold.

Definition 3.1 *We say that an algorithm outputs the correct answer with high probability or* whp *in time $\Theta(f(n))$ if for every $\alpha \geq 1$, there exists a value $c_\alpha > 0$ depending on α, such that after $\Theta(f(n))$ time, the algorithm outputs the correct answer with probability at least $1 - \frac{c_\alpha}{n^\alpha}$. Note that the constant in the Θ notation may also depend on α.*

4 Randomised k-mismatch

The overall strategy is to repeatedly sample single mismatches using a fast solution for the 1-mismatch problem with don't cares. This is achieved by effectively masking out a number of positions in the pattern with don't care symbols, thus making it likely that exactly one of the remaining positions will result in a mismatch. In this way, we can sample single mismatches even when the true Hamming distance may be considerably larger.

We call such a masked version of the pattern a subpattern. A number of subpatterns will be chosen at random and for each one, the 1-mismatch pattern

matching algorithm is performed. Each 1-mismatch operation will tell us the location of a mismatch with the subpattern if one occurs, at each position in the text. We will show that after $\Theta(k \log n)$ iterations, all of the mismatches at each alignment that has up to k mismatches, will have been identified and counted whp. As each 1-mismatch stage takes $\Theta(n \log m)$ time the overall running time of the algorithm is $\Theta(nk \log m \log n)$. We will then show how to reduce the running time further by recursively halving the number of sampling iterations.

1-mismatch

The 1-mismatch problem is to determine in which alignments p and t have exactly one mismatch and to identify the location of the mismatch for each such alignment. More formally we wish to find all i s.t. $HD(i) = 1$ and for each such i, find the unique position i' s.t. $p[i' - i + 1] \neq t[i']$. The method that we employ is to modify Equation 1 to give us the required information. The method is shown in Algorithm 1. For simplicity of notation, here and for the rest of the paper we write Σ_j instead of $\Sigma_{j=1}^{m}$ and p' and t' are defined as in Section 1. $[n]$ is further defined to be the set of integers $\{1, \ldots, n\}$.

Input: Pattern p, text t
Output: $B[i]$ contains mismatch location in t for each alignment where $HD(i) = 1$
Compute array $A_0[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
Compute array $A_1[i] = \sum_j (i + j - 1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
foreach $i \in [n]$ **do**
  if $A_0[i] \neq 0$ **then**
    $B[i] \leftarrow A_1[i]/A_0[i]$;
  else
    $B[i] \leftarrow$ No_Mismatch;
foreach $i \in [n]$ *s.t.* $B[i] \neq$ *No_Mismatch* **do**
  if $(p[B[i] - i + 1] - t[B[i]])^2 \neq A_0[i]$ **then**
    $B[i] \leftarrow$ More_Than_1_Mismatch;

Algorithm 1: 1-mismatch

For any i where there are no mismatches between p and $t[i, \ldots, i + m - 1]$, both $A_0[i] = 0$ and $A_1[i] = 0$. If there is exactly one mismatch then $B[i]$ is its location in t. The check is to ensure that the value of $B[i]$ came from no more than 1 mismatch. The following Lemma gives the correctness and time complexity of the algorithm.

Lemma 4.1 *Algorithm 1 solves the* 1*-mismatch problem in* $\Theta(n \log m)$ *time.*

Proof: For a fixed alignment i of the text with respect to the pattern, there are three cases for Algorithm 1:

1. $HD(i) = 0 \Rightarrow A_0[i] = 0$ and $B[i]$ is correctly set to No_Mismatch.
2. $HD(i) = 1 \Rightarrow$ There is exactly one mismatch at some position ℓ. Therefore $A_0[i] = (p_{\ell-i+1} - t_\ell)^2$ and $A_1[i] = \ell(p_{\ell-i+1} - t_\ell)^2$. Therefore $B[i] = A_1[i]/A_0[i] = \ell$ which gives the location of the mismatch in t.
3. $HD(i) > 1 \Rightarrow (p[B[i]-i+1]-t[B[i]])^2 < A_0[i]$. Therefore $B[i]$ is correctly set to More_Than_1_Mismatch.

The overall running time of Algorithm 1 is dominated by the time taken to perform the FFTs needed in order to compute A_0 and A_1. Therefore the running time is $\Theta(n \log m)$. ■

Sampling and Matching

We now show how Algorithm 1 can be repeatedly applied to random subpatterns of p to solve the full k-mismatch problem. Each subpattern p^* is chosen at random by sampling locations j uniformly at random from the pattern.

The sampling rate is set to $1/k$ so that the average number of selected locations is m/k. A subpattern p^* is created by setting $p^*_j = p_j$ for those chosen locations. We set the characters of all other positions in p^* to be the don't care symbol and ensure $|p^*| = |p|$.

We then run the 1-mismatch algorithm using p^* and t. This whole "sample and match" process is repeated $\Theta(k \log n)$ times, each time keeping record of where any single mismatch occurs for each index i in the text. Algorithm 2 sets out the main steps.

Input: Pattern p, text t and an integer k
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
for *times* $= 1$ *to* $\Theta(k \log n)$ **do**
    `/* Sample and Match stage */`
    Sample subpattern p^* with sample rate $1/k$;
    1-mismatch(p^*, t);
$L[i] \leftarrow$ total number distinct mismatches found at alignment i;
`/* Checking stage */`
Check at each position i in t that all mismatches were found;
$O[i] \leftarrow L[i]$, if all mismatches found, otherwise $O[i] \leftarrow \perp$;

Algorithm 2: Randomised k-mismatch with don't cares

Theorem 4.2 *Algorithm 2 computes $HD_k(i)$ for all locations i whp.*

Proof: We first analyse the algorithm for a single alignment of the pattern and text and show that if there are no more than k mismatches, the Hamming distance is correctly computed whp. By adding a final checking stage, we then show that this probabilistic bound is sufficient to show that $HD_k(i)$ will be correctly computed for all alignments i whp.

Suppose $d = HD(i) \leq k$, and let $i_1, ..., i_d$ be the locations of the mismatches. We focus wlog, on a single mismatch i_1. The sample and match stage will find mismatch i_1 if and only if $p^*[i_j] = \phi$ for all $j > 1$ and $p^*[i_1] = p[i_1]$. The probability of this event happening is $((k-1)/k)^{k-1}/k$ which is bounded below by $1/ek$ for all $k \geq 2$.

Therefore, the probability that $\Theta(k \log n)$ iterations will not find i_1 is at most $(1 - \frac{1}{ek})^{\Theta(k \log n)} \leq n^{-c}$, for some value c which grows linearly with the number of iterations of the sample and match stage. Thus, i_1 will be found whp after $\Theta(k \log n)$ iterations. Hence, according to the union bound, all mismatch positions i_j will also be found whp after $\Theta(k \log n)$ iterations.

It remains to show we can determine if the Hamming distance is in fact greater than k as the algorithm so far relies on the assumption that $HD(i) \leq k$. This is performed by an extra checking stage which checks to see if all the mismatches at a given location have been found. An important feature of Algorithm 1 is that it gives us not only the location ℓ of the error but also the value $A_0[\ell - i + 1] = (p_{\ell - i + 1} - t_\ell)^2$ for each i where there is a 1-mismatch. If we first compute $C[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$ for all i, then we can "correct" this cross-correlation for each distinct 1-mismatch using the value of $A_0[i]$ found during the running of 1-mismatch. For each iteration of 1-mismatch that results in a new mismatch being found at position i, we subtract the value of $A_0[i]$ from $C[i]$. In this way, we can check whether we have found all the mismatches at a given position by keeping track of which mismatches have been seen before, using a binary search tree for example. The time required to perform insertions and lookups in the binary search trees is subsumed by the time required to perform each cross-correlation calculation and so does not affect the overall time complexity of the algorithm.

We now know that if $C[i] \neq 0$ after correcting all the up to k different mismatches found at some position i, then whp there is no k-mismatch at that position. Notice that if exactly k mismatches have been found at position i, then $C[i] \neq 0$ implies with certainty that $HD(i) > k$. ■

The following Theorem gives the running time of the algorithm.

Theorem 4.3 *Algorithm 2 runs in $\Theta(nk \log m \log n)$ time.*

Proof: The algorithm runs $\Theta(k \log n)$ sample and match iterations, each taking $\Theta(n \log m)$ time to compute. After that, it goes over the array $C = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$ and checks for each location i whether the distinct

mismatch contributions sum up to $C[i]$. This is done in $\Theta(n \log m + nk \log k)$ time. Therefore, the overall running time of Algorithm 2 is $\Theta(nk \log m \log n)$. ■

An $\Theta(n(k + \log m \log k) \log n)$ time recursive algorithm

We now show how to improve the time complexity of the previous algorithm from $\Theta(nk \log m \log n)$ to $\Theta(n(k + \log m \log k) \log n)$. The approach is recursive and requires us to halve the number of mismatches we are looking for at each turn. To simplify the explanation consider the worst case, where the number of mismatches at a alignment is at least k. If the number of mismatches is less than k, then the algorithm can only find the mismatches more quickly and so the bounds given still hold.

Our improved algorithm will work in $\log k$ recursive stages. At stage s, the algorithm will whp have found all but at most $k_s = 2^{-s}k$ mismatches in any alignment j assuming $HD(j) \leq k$. We will show how to effectively ignore all previously found mismatches when performing the 1-mismatch algorithm. The main motivation behind this approach is the observation that when sampling, it is the final mismatch that will take the longest to find. However, in our approach, by being able to disregard the contribution of previously found mismatches to the cross-correlations, the final mismatch becomes the easiest to find thus giving us our desired speedup.

Algorithm 3 gives an outline of the recursive k-mismatch algorithm. In this algorithm we will also have to maintain a second data structure, E, containing the same set of previously found distinct mismatches. However, in this case the data structure is to be implemented differently than before. The mismatches are held in an array of lists. The array has size m and list j in this array will contain all mismatches found so far that occur between the text and location j of the pattern, p_j. The total size can not be greater than nk as we only consider at most k mismatches per position in the text. We initialise E to be empty as a first step of the algorithm.

Our solution to the problem of disregarding the contributions of previously found mismatches is to correct the sums in arrays A_0 and A_1 prior to their being used. We show the modified self-correcting 1-mismatch algorithm in Algorithm 4.

We will next show that our overall scheme answers the k-mismatch problem correctly whp. Lemma 4.5 shows that we can find $k_s/2$ of the required mismatches using $\Theta(k_s + \log n)$ executions of the Algorithm 4. The proof of this Lemma employs Theorem 4.4 which is a version of a Chernoff-Hoeffding bound (see e.g. [AS01] for standard versions of these bounds).

Theorem 4.4 (Chernoff bound - error relative to 1st moment)

Input: Pattern p, text t
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
`/* E holds all previously found mismatches */`
Initialise E;
Set $k_0 = k$;
for $s = 0$ *to* $\lfloor \log k \rfloor$ **do**
  for *times* $= 1$ *to* $\Theta(k_s + \log n)$ **do**
    `/* Sample and Match stage */`
    Sample a subpattern p^* with sample rate $1/k_s$ from p;
    Algorithm 4(p^*, t, E);
  Update E according to the mismatches found in these iterations;
  $k_{s+1} = k_s/2$;
Let $L[i]$ = total number distinct mismatches found at alignment i;
`/* Checking stage */`
Check at each position i in t that all mismatches were found;
$O[i] = L[i]$, if all mismatches found, otherwise $O[i] = \perp$;

Algorithm 3: Faster recursive randomised k-mismatch with don't cares

Input: Pattern p, text t, previously discovered mismatches E
Output: Array B, containing the location of the mismatch in each alignment where there is one mismatch not yet discovered
Compute array $A_0[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
Compute array $A_1[i] = \sum_j (i+j-1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
foreach $i \in [n]$ **do**
  Correct $A_0[i]$ and $A_1[i]$ according to E;
  if $A_0[i] \neq 0$ **then**
    $B[i] = A_1[i]/A_0[i]$;
  else
    let $B[i]$ = No_Mismatch;
foreach $i \in [n]$ *s.t.* $B[i] \neq$ *No_Mismatch* **do**
  if $(p[B[i] - i + 1] - t[B[i]])^2 \neq A_0[i]$ **then**
    $B[i]$ = More_Than_1_Mismatch;

Algorithm 4: Self-correcting 1-mismatch

Assume random variables $X_1, ..., X_m$ are i.i.d. and $X_i \in [0,1]$. Let $\mu = E(X_i)$. Then

$$Pr\left(\frac{1}{m}\sum X_i \leq (1-\delta)\mu\right) < e^{-m\mu\delta^2/2}$$

Lemma 4.5 *After $\Theta(k_s + \log n)$ iterations of sample and match stage, the locations and values of at most $k_s/2$ different mismatches will remain to be found whp.*

Proof: We assume for simplicity the worst case where the number of mismatches remaining to be found is k_s. The probability of finding a mismatch after one iteration is at least $1/e$. We apply the Chernoff bound, Theorem 4.4 with random variable $X_i = 1$ if at the ith stage a mismatch is found and 0 otherwise and $\delta = 1/2$, Therefore at least $x/2e$ not necessarily distinct mismatches are found after $x = \Theta(k_s + \log n)$ iterations whp.

On the other hand, fix some set of mismatch locations of size $k_s/2$. The probability that the mismatches found are entirely contained in this set is at most $2^{-x/2e}$. Therefore, the probability that we will have found fewer than $k_s/2$ distinct mismatches overall is at most $2^{-x/2e}\binom{k_s}{k_s/2}$. It follows that given we have found $x/2e$ not necessarily distinct mismatches in total, we will have found at least $k_s/2$ distinct mismatches with probability no less than $1 - 2^{-x/2e}\binom{k_s}{k_s/2}$. Therefore, after $\Theta(k_s + \log n)$ iterations of Algorithm 4, at least $k_s/2$ mismatches will be found whp. ■

In order to show the claimed running time of Algorithm 4, we will have to prove that computing the contribution of previously discovered mismatches at each stage takes $\Theta\left(\frac{nk}{k_s}\right)$ time. Performed naively, this computation would take $\Theta(nk)$ time as for each alignment, there could be as many as k previously discovered mismatches. We rely on the fact that a sampled subpattern is likely not to have more than cm/k_s positions which are not don't cares, for some constant c.

We correct the contributions of previously found mismatches as follows. In order to correct the mismatches of a single sampled subpattern p^*, we need only consider the positions in the array E which relate to positions which do not contain don't cares in p^*. For each such position j in the array, we look up each position i' in the text where a mismatch will occur with position j of the pattern and calculate the contribution they make to $A_0[i' - j + 1]$ and $A_1[i' - j + 1]$ required by Algorithm 4. This can be done in constant time per mismatch. The number of such mismatches is at most nk/k_s in expectation and we will next show, that it is also small whp. We will need another version of a Chernoff-Hoeffding bound (see [AS01] again) which is given in the following Theorem.

Theorem 4.6 (Chernoff bound - error relative to 2nd moment) *Let $X_1, ..., X_m$ be discrete, independent random variables such that $E(X_i) = 0$ and $|X_i| \leq 1$ for all i. Let $X = \sum_{i=1}^{m} X_i$. Then*

$$Pr\left(X \geq \lambda\sqrt{Var(X)}\right) \leq e^{-\lambda^2/4}$$

for any $0 \leq \lambda \leq 2\sqrt{Var(X)}$.

We can now show the total number of not necessarily distinct mismatches that will need to be corrected in one stage of the recursive algorithm.

Lemma 4.7 *If the sample and match stage is run $\Theta(k_s + \log n)$ times, the number of times a previously discovered mismatch is found is $O(nk(k_s + \log n)/k_s)$ whp.*

Proof: Fix the number of iterations of the sample and match stage to be $r = d(k_s + \log n)$ for some constant d. We will show that for d sufficiently large, the claim holds whp. Clearly if the claim is true for large d it will also be true for smaller values of d as the number of mismatches found increases with the number of iterations. We consider the random variables $\{X_{i,j}\}_{i\in[m],j\in[r]}$ which indicate whether p_i was replaced with a ϕ in iteration j. We let $X_{i,j} = 1$ with probability $1/k_s$ and 0 otherwise. Let a_i be the number of mismatches previously found at index i of the pattern when comparing it to the text in all alignments. The total number of these previously found mismatches over all r iterations is therefore $X = \sum_{i\in[m],j\in[r]} a_i X_{i,j}$. We want to show that X is $O(nk(k_s + \log n)/k_s)$ whp.

In order to be able to use the Chernoff bound we need to define $Y_{i,j} = \frac{a_i}{n} X_{i,j}$ and $Y = \sum_{i=1}^{m} Y_i = \frac{1}{n}X$ and show several bounds on these random variables. We notice that $Y_i \in \{0, \frac{a_i}{n}\} \in [0, 1]$. Hence, according to the Chernoff bound (Theorem 4.6) as long as $r < 4Var(Y)$

$$Pr\left(Y - E(Y) \geq \sqrt{rVar(Y)}\right) \leq e^{-r/4} < e^{-d\log n/4} \tag{2}$$

meaning $Y < E(Y) + \sqrt{rVar(Y)}$ whp. Notice though that if $Var(Y)$ is smaller then $r/4$, the probabilities must be even better, and surely $Y < E(Y) + \sqrt{rVar(Y)}$ whp. Next, we calculate $E(Y)$ and $Var(Y)$:

$$E(Y) = \sum_{i\in[m],j\in[r]} \frac{a_i}{n} E(X_{i,j}) = \sum_{i\in[m],j\in[r]} \frac{a_i}{n}\frac{1}{k_s} = \frac{r}{nk_s}\sum_{i=1}^{m} a_i$$

$$Var(Y) = \sum_{i\in[m],j\in[r]} \frac{a_i^2}{n^2} Var(X_{i,j}) \leq \sum_{i\in[m],j\in[r]} \frac{a_i^2}{n^2}\frac{1}{k_s} = \frac{r}{n^2k_s}\sum_{i=1}^{m} a_i^2$$

Finally, the sum $\sum_{i=1}^{m} a_i$ is the number of mismatches previously found, and so, it is bounded above by nk. The number of mismatches in each index is

bounded by n, hence the sum $\sum_{i=1}^{m} a_i^2$ is bounded by $\max a_i \sum_{i=1}^{m} a_i \leq n^2 k$. Therefore it follows that $Var(Y) \leq rk/k_s$.

By applying inequality 2, we get that $Y < \frac{rk}{k_s} + \sqrt{\frac{r^2 k}{k_s}}$ whp, and so, X is $O(\frac{k}{k_s} nr)$ whp as required.

■

Using Lemma 4.7 we can now give the time complexity of running the self-correcting 1-mismatch algorithm $\Theta(k_s + \log n)$ times:

Corollary 4.8 *The time complexity of running the self-correcting* 1*-mismatch algorithm* $\Theta(k_s + \log n)$ *times (as needed in one stage of our recursion) is* $\Theta\left(n\left(k + \frac{k}{k_s}\log n + k_s \log m + \log n \log m\right)\right)$.

Proof: Computing A_0 and A_1 using FFTs, takes $\Theta(n \log m)$ time for each run. Therefore, overall, their computation takes $\Theta(n(k_s + \log n)\log m)$ time. The time complexity of handling all the previously found mismatches in all runs, accumulates to $\Theta(kn(k_s + \log n)/k_s)$ according to Lemma 4.7.

Therefore, the overall time complexity is $\Theta(n(k + k/k_s \log n + k_s \log m + \log n \log m))$. ■

We can now give the final time complexity for the recursive randomised k-mismatch with don't cares algorithm.

Theorem 4.9 *Algorithm 3 runs in* $\Theta(n(k + \log m \log k)\log n)$ *time.*

Proof: Let us concentrate on the i-th stage of the recursion. At this stage we need to solve the $k/2^i$-mismatch problem, by running the self-correcting 1-mismatch algorithm $\Theta\left(2^i + \log n\right)$ times. According to Corollary 4.8 this takes overall $\Theta\left(n\left(k + \frac{k}{2^i}\log n + 2^i \log m + \log n \log m\right)\right)$ time.

Summing up the time taken by all stages together we therefore get $\Theta(n(k \log k + k \log n + k \log m + \log n \log m \log k))$, which can be simplifed to $\Theta(n(k + \log m \log k)\log n)$ ■

5 Deterministic k-mismatch with don't cares

In this Section we give an $\Theta\left(nk^2 \log^2 m\right)$ time deterministic solution for the k-mismatch with don't cares problem. Our algorithm is based on Algorithm 2 but instead of simply taking random subpatterns of our pattern, it uses a group testing scheme to select the subpatterns.

Group testing is a long studied problem in combinatorics. A small set of k ill people are to be identified from a set of size m by using only queries (tests) of the form "Does set X contain an ill person?". Group testing is used in many applications, including detecting syphilis or HIV in blood samples [Dor43],

quality control [SG59], communication [KS64, Wol85], software testing [BG02, CDFP97] and numerous examples in computational molecular biology [DH00, FKKM97, ND00, BBKT96, PL94, BLC91, BBK$^+$95, TJP00]. The conventional use of group testing is as a stand alone tool but in this paper, we use it in a novel way as a building block of our algorithm.

The definition of a group testing scheme (GT) is as follows:

Definition 5.1 *Consider a universe U of size m. A family of tests (subsets) $\mathcal{F} \subset \mathcal{P}(U)$ is a group testing scheme of strength r ((m,r)-GT) if for any subset $A \subset U$ of size at most r, and for any element $x \notin A$, there exists a test $B \in \mathcal{F}$ that distinguishes x from A, meaning $x \in B$ while $A \cap B = \varnothing$.*

Our algorithm, as other applications of group testing, requires a group testing scheme which is as small as possible. The smallest explicit construction for (m,r)-GT schemes hitherto contains $t = \Theta\left(\min[r^2 \ln m, m]\right)$ tests, and takes $\Theta\left(rm \ln m\right)$ time to build [PR08].

Theorem 5.2 ([PR08]) *Let m and r be positive integers. It is possible to construct an (m,r)-GT scheme containing $\Theta\left(\min[r^2 \ln m, m]\right)$ tests in $\Theta\left(rm \ln m\right)$ time.*

Our usage of group testing is straightforward using the combinatorial concepts of *selection by intersection* and *strongly-selective families* (SSF) [CMS03]. Selection by intersection means distinguishing an element from a set of elements by intersecting it with another set. More precisely:

Definition 5.3 *Given a subset $A \subset U$ of a universe U, element $x \in A$ is* selected *by subset $B \subset U$ if $A \cap B = \{x\}$. An element is* selected *by a family of subsets $\mathcal{F} \subset \mathcal{P}(U)$ if one of the subsets in $\mathcal{F}$ selects it.*

An SSF is a family of subsets that select any element out of a small enough subset of the universe. More precisely,

Definition 5.4 *A family $\mathcal{F} \subset \mathcal{P}(U)$ is said to be* (m,r)-strongly-selective *if, for every subset $A \subset U$ of size $|A| = r$, all elements of A are selected by $\mathcal{F}$. We call such a family an (m,r)-SSF.*

SSFs and group testing schemes are almost equivalent. On the one hand, an $(m, r+1)$-SSF is a group testing scheme of strength r. On the other hand, a group testing scheme of strength r in a universe of size m is an (m,r)-SSF. For a detailed proof see [KS64]. We will work with the notation of SSFs from now on. Rewriting Theorem 5.2 in terms of SSFs we get the following result.

Corollary 5.5 *Let m and r be positive integers. It is possible to construct an (m,r)-SSF of size $\Theta\left(\min[r^2 \ln m, m]\right)$ in $\Theta\left(rm \ln m\right)$ time.*

How will we choose the subpatterns p^*? We give a 1-1 correspondence between the patterns p^* and the sets in the SSF. Each test of the SSF can be regarded as a set of locations in the input pattern p. We use these locations to form the subpatterns p^* so that $p^*_j = p_j$ for the locations that are in the test and $p^* = \phi$, the don't care symbol, otherwise.

An overview of the deterministic solution for k-mismatch with don't cares algorithm is given in Algorithm 5. Note that the strength r is now set to k and that this deterministic algorithm is similar in structure to the randomised solution presented in Algorithm 2.

Input: Pattern p, text t and an integer k
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
foreach $s \in (m,k)$-*SSF* **do**
    `/* Sample and Match stage */`
    Choose subpattern p^* according to s;
    Algorithm 1(p^*, t);
Let $L[i]$ = total number distinct mismatches found at alignment i ;
`/* Checking stage */`
Check at each position i in t that all mismatches were found;
$O[i] = L[i]$, if all mismatches found, otherwise $O[i] = \perp$;

Algorithm 5: Deterministic k-mismatch with don't cares

We can now show that Algorithm 5 gives a deterministic solution for the k-mismatch problem with don't cares.

Theorem 5.6 *Algorithm 5 computes $HD_k(i)$ for all locations i.*

Proof: Let i be an alignment of the pattern with respect to the text for which $\ell = HD(i) \leq k$. Let $j_1, ..., j_\ell \in [m]$ be the mismatch locations in the pattern. We show wlog that the mismatch in location j_1 will be found. According to the definition of an (m,k)-SSF, j_1, is selected by at least one of the tests in the family. The subpattern which matches this set will have don't cares in all locations j_h except in j_1. Therefore, this subpattern will have only one mismatch with the text in alignment i. Hence, location j_1 would be found when running the Algorithm 1 on p^* and t as required.

We can now see that for all alignments i where $HD_k(i) \leq i$, all mismatches will be found. Moreover, the checking phase will detect if the Hamming distance is in fact greater than k. It therefore follows that Algorithm 5 provides a full deterministic solution to the k-mismatch problem with don't cares. ■

The running time is given in the following Theorem.

Theorem 5.7 *Algorithm 5 runs in $\Theta\left(nk^2 \log^2 m\right)$ time.*

Proof: The (m,k)-SSF can be computed in $\Theta(mk \log m)$ time, creating $\Theta\left(k^2 \log m\right)$ tests. For each subpattern, the 1-mismatch calculation takes $\Theta(n \log m)$ time or $\Theta\left(nk^2 \log^2 m\right)$ time overall. Storing and counting the up to k distinct 1-mismatches at each location will take $\Theta(nk \log k)$ further time. Finally, we can check and correct all mismatches found in $\Theta(n \log m + nk)$ time. The total running time is therefore $\Theta\left(nk^2 \log^2 m\right)$ as required. ■

Further deterministic speedups

So far we have based our deterministic algorithm on the randomised Algorithm 2, using group testing schemes instead of randomness. With some more effort, we can base the deterministic algorithm on the faster randomised Algorithm 3. However in order to benefit from this, we need to also replace the usage of group testing schemes with another derandomisation tool. In the analysis of Algorithm 3 the speed advantage came from the observation that finding the first $k/2$ mismatches is easier than finding the last $k/2$. We therefore need a derandomisation tool to help us exploit this fact and group testing will not suffice on its own. Fortunately, there exists a tool suitable for our needs which is known as a selector. An (m,k)-selector is similar to a (m,k)-SSF, but instead of selecting all k elements as the SSF would do, it instead only guarantees to select at least $k/2$ of them. This weaker property of selectors has the advantage that the lower bound for the number of tests for a selector is $\Omega(k \log_k m)$ whereas it is $\Omega(k^2 \log_k m)$ for group testing [CMS01]. Therefore, using selectors instead of SSFs could potentially save a factor of k in the running time. Unfortunately, there are no known efficient algorithms for explicitly building selectors of size smaller then $\Theta(k \text{polylog} m)$ [CK05] where polylog$m$ hides a large exponent. We finish with the following two remarks.

Remark 1 *The k-mismatch problem with don't cares can be solved using (m,k)-selectors instead of (m,k)-SSFs combined with Algorithm 3.*

Remark 2 *As the best construction of (m,k)-selectors is of size $\Theta(k \text{polylog} m)$ and takes $\Theta(m \text{polylog} m)$ time to construct, the new algorithm will then run in $\Theta(nk \text{polylog} m)$. However, more efficient constructions of selectors will translate into more efficient algorithms for the k-mismatch problem.*

6 Conclusion and Open Problems

We have presented the first non-trivial algorithms for the k-mismatch problem with don't cares. We conjecture that the gap between the deterministic and randomised complexities can be closed. A further interesting open question is whether an $\tilde{\Theta}(n\sqrt{k})$ algorithm can be found to match the fastest known solution for the problem without don't care symbols.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM journal on Computing*, 16(6):1039–1051, 1987.

[Aku95] T. Akutsu. Approximate string matching with don't care characters. *Information Procesing Letters*, 55:235–239, 1995.

[ALP04] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

[AS01] N. Alon and J. Spencer. *The Probabilistic Method*. John Wiley and Sons Inc., 2nd edition, 2001.

[BBK+95] W.J. Bruno, D.J. Balding, E. Knill, D. Bruce, C. Whittaker, N. Dogget, R. Stalling, and D.C. Torney. Design of efficient pooling experiments. *Genomics*, 26:21–30, 1995.

[BBKT96] D. J. Balding, W. J. Bruno, E. Knill, and D. C. Torney. A comparative survey of non-adaptive pooling designs. *Institute for Mathematics and Its Applications*, 81:133–+, 1996.

[BG02] A. Blass and Y. Gurevich. Pairwise testing. *Bulletin of the EATCS*, 78:100–132, 2002.

[BLC91] E. Barillot, B. Lacroix, and D. Cohen. Theoretical analysis of library screening using a N-dimensional pooling strategy. *Nucleic Acids Research*, 19(22):6241–6247, 1991.

[BM77] R. S. Boyer and J. S. Moore. A fast string matching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[CC07] P. Clifford and R. Clifford. Simple deterministic wildcard matching. *Information Processing Letters*, 101(2):53–54, 2007.

[CDFP97] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The AETG system: An approach to testing based on combinatiorial design. *Software Engineering*, 23(7):437–444, 1997.

[CGL04] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC '04: Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 91–100, 2004.

[CH02] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC '02: Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 592–601, 2002.

[CK05] B. S. Chlebus and D. R. Kowalski. Almost optimal explicit selectors. In Maciej Liskiewicz and Rüdiger Reischuk, editors, *Proceedings of Fundamentals of Computation Theory (FCT 2005)*, volume 3623 of *LNCS*, pages 270–280. Springer, 2005.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CLS+06] Ho-Leung Chan, Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. In *CPM*, pages 49–59, 2006.

[CMS01] Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Selective families, superimposed codes, and broadcasting on unknown radio networks. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms (SODA '01)*, pages 709–718, 2001.

[CMS03] A.E.F. Clementi, A. Monti, and R. Silvestri. Distributed broadcast in radio networds of unknown topology. *Theoretical Computer Science*, 302(1–3):337–364, 2003.

[CP07] R. Clifford and E. Porat. A filtering algorithm for k-mismatch with don't cares. In *14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 130–136, October 2007.

[DH00] D. Z. Du and F. K. Hwang. *Combinatorial Group Testing and its Applications*, volume 12 of *Series on Applied Mathematics*. World Scientific, 2nd edition, 2000.

[Dor43] R. Dorfman. The detection of defective members of large populations. *The Annals of Mathematical Statistics*, 14(4):436–440, 1943.

[FKKM97] M. Farach, S. Kannan, E. Knill, and S. Muthukrishnan. Group testing problems with sequences in experimental molecular biology. In *the Compression and Complexity of Sequences 1997*, page 357, 1997.

[FP74] M. Fischer and M. Paterson. String matching and other products. In R. Karp, editor, *Proceedings of the 7th SIAM-AMS Complexity of Computation*, pages 113–125, 1974.

[Ind98] P. Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 166–173, 1998.

[Kal02] A. Kalai. Efficient pattern-matching with don't cares. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 655–656, 2002.

[Kar93] H. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

[KMP77] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[Kos87] S. R. Kosaraju. Efficient string matching. Manuscript, 1987.

[KS64] W.H. Kautz and R.C. Singleton. Nonrandom binary superimposed codes. *IEEE Transaction of InformationTheory*, 10:363–377, 1964.

[LV85] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *FOCS '85: Proceedings of the 26th Symposium on Foundations of Computer Science*, pages 126–136, 1985.

[LV86] G. M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[ND00] H.Q. Ngo and D.Z. Du. A survey on combinatorial group testing algorithms with applications to DNA library screening. In *DIMACS Series Discrete Math. and Theor. Computer Science 55, AMS 2000*, pages 171–182, 2000.

[PL94] Pavel A. Pevzner and Robert J. Lipshutz. Towards dna sequencing chips. In *MFCS '94: Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science 1994*, pages 143–158, London, UK, 1994. Springer-Verlag.

[PR08] E. Porat and A. Rothschild. Explicit non-adaptive combinatorial group testing schemes. In *Automata, Languages and Programming: 35st International Colloquium, ICALP 2008*, 2008.

[SG59] M. Sobel and P.A. Groll. Group testing to eliminate efficiently all defectives in a binomial sample. *Bell Syst. Tech. J.*, 38:1179–1252, 1959.

[TJP00] Berger T., Mandell J.W., and Subrahmanya P. Maximally efficient two-stage screening. *Biometrics*, 56:833–840(8), September 2000.

[Wol85] J.K. Wolf. Born again group testing: Multiaccess communications. *IEEE Transactions on Information Theory*, 31(2):185–191, 1985.