A filtering algorithm for k-mismatch with don't cares

Raphaël Clifford[1] and Ely Porat[2]

[1] University of Bristol, Dept. of Computer Science, Bristol, BS8 1UB, UK
clifford@cs.bris.ac.uk
[2] Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel
porately@cs.biu.ac.il

Abstract. We present a filtering based algorithm 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 in either p or t (but not both), and a bound k, our algorithm finds all the places that the pattern matches the text with at most k mismatches. The algorithm is deterministic and runs in $\Theta(nm^{1/3}k^{1/3}\log^{2/3} m)$ time.

1 Introduction

We consider approximate string matching under the widely used Hamming distance. In particular our interest is in 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, we allow either the pattern or the text (but not both) to contain promiscuously matching don't care symbols. Given a bound k, our algorithm finds all the locations where 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 exact string matching is a classic one in computer science whose linear time solutions were first presented in the 1970s [4, 15]. Determining the time complexity of exact matching with optional single character *don't care* symbols has also been well studied. Fischer and Paterson [12] presented the first solution based on fast Fourier transforms (FFT) with an $\Theta(n \log m \log |\Sigma|)$ time algorithm in 1974[3], 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 [13] 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 [14]. In 2002, the first deterministic $\Theta(n \log m)$ time solution was given [9] which was then further simplified in [7].

The key observation given by [7] but implicit in previous work is that for numeric strings, if there are no don't care symbols, then for all locations $1 \leq i \leq n - m + 1$ we can calculate

[3] Throughout this paper we assume the word 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.

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

in a total of $\Theta(n \log m)$ time using FFTs. Here p_j indicates the jth symbol in the input string p. The notation holds similarly for t_i in string t. 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 also be computed in $\Theta(n \log m)$ time using FFTs.

Approximate matching is one of the fundamental tools of large scale data processing and is both widely studied and used in practice. Errors or noise can occur in data in a great variety of forms and so any search tool on real data must be able to handle a level of approximation. In other cases the data themselves are accurate but we might be tasked for example, with searching for similarity between images in a library or in the case of bioinformatics, a key operation is to search for functional similarities between genes or proteins. In some cases elements of the data are simply unknown and should not be counted towards any measure of similarity. Such data are often represented by don't care or wildcard symbols. As an example in image processing, a rectangular image segment may contain a facial image and the objective is to identify the face in a larger scene. However, background pixels around the faces may be considered to be irrelevant for facial recognition and these should not affect the search algorithm. Alternatively, a consensus sequence derived from multiple alignment in computational biology (see e.g. [11]) may contain unknown values and the aim is to perform approximate matching rapidly on a large DNA or protein database without penalising any matches to the unknown values. Due to the asymmetry between query and database or pattern and text it is often the case that uncertainty lies in either the pattern or the text but not both. It is this model of approximate matching with don't cares in either the pattern or text that we consider here.

In Section 2 related and previous work is discussed. We then formalise the pattern matching problem description and give basic definitions that will be used throughout in Section 3. In Section 4 we explain in detail why don't cares cause problems for traditional filtering algorithms and present our main solution. Finally, in Section 5 we conclude and discuss the open problems that remain to be solved.

2 Related and previous work

Much progress has been made in finding fast algorithms for the k-mismatch problem *without* don't cares over the last 20 years. $\Theta(n\sqrt{m \log m})$ time solutions to the k-mismatch problem based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [1, 16]. 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 LCA operations on the suffix tree of p and t [18]. This was subsequently improved to $\Theta(n\sqrt{k \log k})$ time by a method based on filtering and FFTs again [3]. Approximations within a multiplicative factor of $(1 + \epsilon)$ to the Hamming distance can also be found in $\Theta(n/\epsilon^2 \log m)$ time [13]. A variant of the edit-distance problem (see e.g. [17]) called the k-difference problem with don't cares was considered in [2]. Progress has also been made recently on the related problem of indexing with errors and don't cares [8, 5].

To the authors' knowledge, no non-naive algorithms have been given to date for the k-mismatch pattern matching problem with don't cares. However, the $\Theta(n\sqrt{m \log m})$ divide and conquer algorithms of Kosaraju and Abrahamson [1, 16] can be easily extended to handle don't cares in both the pattern and text without changing the overall time complexity. This is because the algorithm counts matches and not mismatches. First we count the number of non-don't care matches in $\Theta(n\sqrt{m \log m})$ time. Then we need only subtract this number from the maximum possible number of non-don't care matches at a particular position in the text in order to count the mismatches. When don't cares are only allowed in only one of the pattern or text this can be computed in linear time as we will show in Section 4.

3 Problem definition and 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. Either the pattern or the text may also include ϕ in their alphabet but not both. 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 the *position*. The term *alignment* will only be used to refer to a position in the text.

- Define $HD(i)$ to be the number of mismatches or 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 $HD_k(i) = \begin{cases} HD(i) \text{ if } HD(i) \leq k \\ \perp \qquad\qquad \text{otherwise} \end{cases}$
- We say that at position i in t, p is a k-mismatch if $HD_k(i) \neq \perp$.
- Symbols other than don't cares are said to be *solid*. We say that a match between two solid symbols is a *solid match* and we say that a position in the pattern or text corresponding to a solid symbol is a *solid position*.

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,

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

can be calculated accurately and efficiently in $\Theta(n \log n)$ time (see e.g. [10], Chapter 32). By a standard trick of splitting the text into overlapping substrings of length $2m$, the running time can be further reduced to $\Theta(n \log m)$.

4 Filtering for the k-mismatch with don't cares problem

Filtering as a technique for speeding up approximate string matching was first introduced in 1990 [6] and has been an active topic of research ever since. The overall aim is quickly to eliminate parts of the text where no match can occur, enabling a slower verification stage to be performed on the remaining candidate positions. Filtering algorithms have however almost exclusively concentrated on improving the average case performance of matching whereas our interest is in improving the worst case. For a comprehensive historical survey of filtering algorithms and approximate matching in general we refer the interested reader to [19].

Our application of filtering will require us to count the number of solid matches between the pattern and substrings of the text. When no don't cares occur in the input there is a simple relationship between the number of matches found and the number of mismatches. This is because the maximum possible number of matches is simply the length of the pattern. When don't cares are introduced into the pattern alone the maximum possible number of solid matches at any given alignment in the text is the number of solid characters in the pattern. However, if don't cares occur in the text then we must explicitly count the number of solid symbols for every alignment. This will correspond to the maximum possible number of solid matches for any query pattern. Fortunately this can also be computed in a straightforward manner in linear time.

Observation: *Given a text t which may contain don't care symbols, the maximum possible number of solid matches between any pattern of length m and the text substrings $t[i, \ldots, i+m-1]$ can be computed in $\Theta(n)$ time.*

Proof. The maximum possible number of solid matches at a given alignment i equals the number of solid characters in the substring $t[i, \ldots, i+m-1]$. To count this for every i, run a sliding window of length m along the text, counting the number of solid characters in each window. After counting the number of solid characters for the first sliding window position in $\Theta(m)$ time, each subsequent count takes constant time to compute by inspecting only two positions in the text. □

We can therefore assume that the maximum possible number of solid matches is known for every alignment of the pattern in the text as it can be precomputed

in linear time. This will allow us to count matches rather than mismatches. We now explain two simple methods which count solid matches between the pattern and text assuming particular conditions on the input. The first is a small modification of a now folklore algorithm dating back at least to the 1970s and the second, although seemingly naive, will lead us to the key filtering result later on.

Algorithm 1: If the alphabet size of the pattern is small, then an $\Theta(|\Sigma|n \log m)$ time pattern matching algorithm will solve the k-mismatch problem with don't cares efficiently. Each solid letter that occurs anywhere in the pattern is considered separately one after the other. A mask is created by replacing all the instances of the chosen symbol in the pattern and text by a 1 and all other symbols (including the don't care symbol) by a 0. For a given symbol x, call the binary modified strings p'_x and t'_x . $p'_x \otimes t'_x$ will give the number of times the symbol x in the pattern matches against itself in the text for every alignment of the pattern and text. As $p'_x \otimes t'_x$ can be computed in $\Theta(n \log m)$ time using FFTs, the total number of solid matches can be calculated for every alignment in $\Theta(|\Sigma|n \log m)$ time by adding the results for each symbol.

Algorithm 2: If no solid symbol in the pattern occurs frequently, then a naive algorithm will count the number of solid matches at each alignment efficiently. First, for each symbol, we store a list of positions where it occurs in p. Then, for each solid position i in the text we can find all the solid positions j in the pattern such that $p_j = t_i$ and add 1 to the $i - j + 1$th position of an auxiliary array C. At the end of this step C will contain the number of solid matches for every location i in t. For example, if $p = ab\phi cd$ and $t = acddd$ then the first position of the auxililiary array C will be set to 2, as $p[1] = t[1]$ and $p[5] = t[5]$. The number of mismatches at the first and only alignment of p and t is therefore $4 - 2 = 2$. The time required for this counting algorithm depends only on how often a frequent symbol is permitted to occur.

It is clear that when the alphabet size is large Algorithm 1 is inefficient and when some symbols occur frequently Algorithm 2 becomes inefficient. Previous filtering approaches have tackled both these situations, however we now show that these methods no longer apply when don't cares can occur in the input.

Why don't cares cause problems for filtering algorithms

The algorithm we present in this paper can be viewed as both a simplification and a generalisation of the approach taken recently in [3] for solving the k-mismatch pattern matching without don't cares. In order to understand the challenges that arise when don't cares are introduced and how we overcome them, we give a brief overview of this previous result. Our presentation is also a considerable simplification of the original, which contained a number of unnecessary complications. In the first step, the symbols in the alphabet are either labelled *frequent* or *infrequent* depending on how many times they occur in the pattern.

1. If there are many frequent symbols then a filter is created from the frequent symbols and used to eliminate positions which could not result in a k-mismatch. The filter is a copy of the pattern with some of the symbols set to the don't care symbol. A verification stage is then run on the remaining positions to confirm if there is a k-mismatch or not. This verification stage takes $\Theta(k)$ time per position that needs to be checked and relies the ability to compute the longest common extension (LCE) between any suffix of the pattern and text in constant time. The LCE is in turn classically implemented using constant time lowest common ancestor (LCA) operations on the generalised suffix tree of p and t [18].
2. If there are few frequent symbols then Algorithm 1 described above is used to count the matches for the frequent symbols. Algorithm 2 can now count the matches for the infrequent symbols.

The main difficulty that arises in this approach when considering don't care symbols is that even assuming that only one of the pattern and text contains don't care symbols, there is no known constant (or even sublinear) time LCE (or LCA) algorithm for strings that allows don't care symbols. If we were to verify each candidate match naively this would require $\Theta(m)$ time per location. As we cannot afford this cost, a different approach is required.

The new filtering algorithm

In order to tackle the k-mismatch problem with don't cares in either the pattern or the text we generalise both the frequent/infrequent approach and provide a novel solution to the problem of verifying candidate matches. We define a solid symbol to be *frequent* if it occurs in the pattern at least d times. We will separate our algorithm into two cases as before. The first case will be when we have fewer than f frequent symbols and the second case is when we have at least f frequent symbols. f and d are variables that we will set later on in order to minimise the overall running time of the algorithm. The two cases are handled as follows.

Case 1: Fewer than f frequent symbols in the pattern

Instead of directly counting the number of mismatches we will count the number of matches of solid symbols and subtract this number from the maximum possible.

1. Count the number of solid matches involving frequent characters. As there are fewer than f frequent characters this can be done with fewer than f cross-correlation calculations following Algorithm 1. The time for this step is therefore $\Theta(nf \log m)$.
2. Count the number of solid matches involving infrequent characters using Algorithm 2. There can be no more than d matching positions for each location in the text and so the total time for this step is $\Theta(nd)$.

The overall running time when there are fewer than f frequent symbols is therefore $\Theta(nd + nf \log m)$.

Case 2: At least f frequent symbols in pattern

When there are at least f frequent symbols we can no longer afford to perform a cross-correlation for every different symbol. Instead we must perform a filtering step to reduce the number of candidate positions in the text at which a k-mismatch can occur. The following Theorem is the basis for the filtering method.

Theorem 1. *Assume there are at least f frequent symbols in the pattern, each of which occurs at least d times. If $fd > k$ then there are at most $\frac{nd}{fd-k}$ possible k-mismatch positions in the text.*

Proof. The first step is to construct an appropriate filter F and then show its use implies the result. For each of f different frequent symbols in p we choose d different locations where they occur in p. At each one of those locations j set $F_j = p_j$. We let $|F| = m$ and place don't care symbols at all other positions in F. In this way there are fd symbols from the alphabet in F and the remaining locations contain don't care symbols which match any symbol in t. For example, to construct a filter for pattern $p = abcb\phi aab$ and $f = d = 2$ we need only take the first two instances the symbols a and b giving the filter $F = ab\phi b\phi a\phi\phi$.

For each position i in t, look up the position of the at most d solid symbols in F that match $t[i]$. The location of each of these matches in the pattern is used to update an auxiliary array C of counts. Specifically, add 1 to each position $C[i - j + 1]$ where it is found that $F[j] = t[i]$.

The sum of all the counts in the array C can be no more than nd as we have added 1 at most d times for each position in t. However, for any given alignment of F in t, the total number of solid matches is at most fd. This implies that if there is a k-mismatch at any position then the corresponding count in C must be at least $fd - k$. Therefore, the total number of alignments for which there is a k-mismatch of the filter F is at most $nd/(fd-k)$. As the filter is a subpattern of p this implies the same upper bound for the number of k-mismatches of p in t. □

Verification in the presence of don't cares

In order to verify whether there are indeed k-mismatches at the candidate positions we split the pattern into ℓ contiguous subpatterns of equal length. If m is not an exact multiple of ℓ then we can pad one end of the pattern with extra don't care characters. For each subpattern of length m/ℓ we run an exact matching with don't cares algorithm on the text and record the locations where a subpattern failed to match exactly. This takes $\Theta(n\ell \log m)$ time in total using for example the algorithm of [7]. Now, for each possible alignment of the pattern in the text, we can check the record of the failed matches of subpatterns and in particular, work out if a pattern aligned at that position is already guaranteed to contain more than than k mismatches. Therefore, for every position i in the text where a k-mismatch can have occurred, we need only associate up to k subpatterns which must contain all the mismatches between p and $t[i, \ldots, i+m-1]$. Any position that would be associated with more than k subpatterns containing

Input: Pattern p, text t and an integer k
Output: $O[i] = HD_k(p, t[i, \ldots, i+m-1])$
if *number of frequent symbols less than threshold* f **then**
    Count "frequent" matches by performing cross-correlations;
    Count "infrequent" matches naively;
    $O[i]$ = max possible number of matches at i minus sum of frequent and infrequent counts;
else
    Construct filter pattern F;
    Eliminate positions that can not result in k-mismatch using F;
    Split pattern into ℓ contiguous subpatterns P_j;
    At each remaining position, eliminate P_j which match exactly;
    $O[i]$ = sum of mismatches for each remaining P_j;
end

Algorithm 3: k-mismatch with don't cares

mismatches can be eliminated. As there are now a total number of nk or fewer subpatterns that contain all relevant mismatches, we can now check each naively to find out how many are in fact associated with each one. The whole process is set out in Algorithm 3

The verification stage is called only for the at most $nd/(fd-k)$ alignments where a k-mismatch can occur. After discarding the subpatterns that match exactly, each alignment now takes $\Theta(km/\ell)$ time to check naively giving an overall time of $\Theta(ndkm/(\ell(fd-k))) + n\ell \log m))$ when there are at least f frequent symbols.

The overall time complexity now depends on setting f, d and l appropriately.

Theorem 2. *The k-mismatch with don't cares problem can be solved using Algorithm 3 in $\Theta(nm^{1/3}k^{1/3}\log^{2/3} m)$ time.*

Proof. The running time of Algorithm 3 is the maximum of the two cases described. This gives the running time as $\Theta(\max(n\ell \log m + ndkm/(\ell(fd-k)), nd + nf \log m))$. In order to minimise this function of three variables we set d to be $\Theta(\sqrt{k \log m})$, ℓ to be $\Theta(\sqrt[3]{\frac{mk}{\log m}})$ and f to be $\Theta(k/d + \sqrt{km/\ell \log m})$. This gives an overall time complexity of $\Theta(nm^{1/3}k^{1/3}\log^{2/3} m)$ as required. □

5 Discussion

We have given the first filtering based algorithm for the k-mismatch problem when don't cares are allowed in either the pattern or the text. There is still a considerable gap between the fastest $\tilde{O}(n\sqrt{k})$ time k-mismatch algorithm without don't cares and our $\tilde{O}(nm^{1/3}k^{1/3})$ time solution which we conjecture can be at least partially closed. An even wider gap exists when don't cares are allowed in both the pattern and text. In this case filtering approaches do not seem to be applicable.

References

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

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

[3] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.

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

[5] H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. A linear size index for approximate pattern matching. In *CPM '06: Proc. 17th Annual Symposium on Combinatorial Pattern Matching*, pages 49–59, 2006.

[6] W. I. Chang and E. L. Lawler. Sublinear expected time approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.

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

[8] R. Cole, L. A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC '04: Proc. 36th Annual ACM Symp. Theory of Computing*, pages 91–100, 2004.

[9] R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC '02: Proc. 34th Annual ACM Symp. Theory of Computing*, pages 592–601, 2002.

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

[11] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1999.

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

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

[14] A. Kalai. Efficient pattern-matching with don't cares. In *SODA '02: Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 655–656, 2002.

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

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

[17] G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. pages 126–136, 1985.

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

[19] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.