

# A Black Box for Online Approximate Pattern Matching

Raphaël Clifford<sup>1</sup>, Klim Efremenko<sup>2</sup>, Benny Porat<sup>3</sup>, and Ely Porat<sup>3</sup>

<sup>1</sup> University of Bristol, Dept. of Computer Science, Bristol, BS8 1UB, UK  
clifford@cs.bris.ac.uk

<sup>2</sup> Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan and Weizman Institute, Dept. of Computer Science and Applied Mathematics, Rehovot, Israel  
klimefrem@gmail.com

<sup>3</sup> Bar-Ilan University, Dept. of Computer Science, 52900 Ramat-Gan, Israel  
bennyporat@gmail.com, porately@cs.biu.ac.il

**Abstract.** We present a deterministic black box solution for online approximate matching. Given a pattern of length  $m$  and a streaming text of length  $n$  that arrives one character at a time, the task is to report the distance between the pattern and a sliding window of the text as soon as the new character arrives. Our solution requires  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1})/n)$  time for each input character, where  $T(n, m)$  is the total running time of the best offline algorithm. The types of approximation that are supported include exact matching with wildcards, matching under the Hamming norm, approximating the Hamming norm,  $k$ -mismatch and numerical measures such as the  $L_2$  and  $L_1$  norms. For these examples, the resulting online algorithms take  $O(\log^2 m)$ ,  $O(\sqrt{m \log m})$ ,  $O(\log^2 m/\epsilon^2)$ ,  $O(\sqrt{k \log k \log m})$ ,  $O(\log^2 m)$  and  $O(\sqrt{m \log m})$  time per character respectively. The space overhead is  $O(m)$  which we show is optimal.

## 1 Introduction

Fast approximate string matching is a central problem of modern data intensive applications. Its applications are many and varied, from computational biology and large scale web searching to searching multimedia databases and digital libraries. As a result, string matching has to continuously adapt itself to the problem at hand. Simultaneously, the need for asymptotically fast algorithms grows every year with the explosion of data available in digital form.

A great deal of progress has been made in finding fast algorithms for a variety of important forms of approximate matching. One of the most studied is the Hamming distance which measures the number of mismatches between two strings. Given a text  $t$  of length  $n$  and a pattern  $p$  of length  $m$ , the task is to report the Hamming distance at every possible alignment.  $O(n\sqrt{m \log m})$  time solutions based on repeated applications of the FFT were given independently by both Abrahamson and Kosaraju in 1987 [1, 17]. Particular interest has been paid to a bounded version of this problem called the  $k$ -mismatch problem. Here a bound  $k$  is given and we need only report the Hamming distance if it is less

than or equal to  $k$ . If the number of mismatches is greater than the bound, the algorithm need only report that fact and not give the actual Hamming distance. In 1985 Landau and Vishkin gave a beautiful  $O(nk)$  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  $O(n\sqrt{k\log k})$  time by a method based on filtering and FFTs again [4]. Approximations within a multiplicative factor of  $(1 + \epsilon)$  to the Hamming distance can also be found in  $O(n/\epsilon^2 \log m)$  time [15].

The problem of determining the time complexity of exact matching with don't cares has also been well studied over many years [13, 15, 16, 12, 6], culminating in two related deterministic  $O(n \log m)$  time solutions. This has been accompanied by recent advances for the problem of  $k$ -mismatch problem with don't cares [10, 9] as well as a surge in interest in provably fast algorithms for distance calculation and approximate matching between numerical strings. Many different metrics have been considered, with for example the  $L_1$  distance [5, 7, 3] and less-than matching [2] problems both being solvable in  $O(n\sqrt{m\log m})$  time and a bounded version of the  $L_\infty$  norm which was first discussed in [8] and then improved in [7, 19] requiring  $O(\delta n \log m)$  time.

In almost every one of these cases and in many others beside, the algorithms make extensive use of the fast Fourier transform (FFT). The property of the FFT that is required 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  $O(n \log n)$  time (see e.g. [11], Chapter 32). By a standard trick of splitting the text into overlapping substrings of length  $2m$ , the running time can be further reduced to  $O(n \log m)$ .

Although the FFT is a very powerful and successful tool, it also brings with it a number of disadvantages. Perhaps most significant of these in this context is that the cross-correlation computation using the FFT is very much an offline algorithm. It requires the entire pattern and text to be available before any search can be performed. Of course, it is not only the FFT that causes this difficulty. For example, the only fast algorithm for the  $k$ -mismatch algorithm which does not employ the FFT uses constant time LCA queries [18]. As it is not known how to perform the necessary preprocessing of a suffix tree to compute the LCA online, this algorithm suffers from the same limitations as those that depend on the FFT.

In many situations such as when monitoring Internet traffic or telecommunications networks this model of computation may not be feasible. It is not sufficient simply that a pattern matching algorithm runs fast. It should also require considerably less space than the input and update at least as quickly as the new data are arriving while still maintaining an overall time complexity which is as close as possible to the full offline algorithm. One approach to handle this situation is the data streaming model where it is assumed that it is not possible to ever store all the data seen and in some variants that only one pass over the data is ever allowed. This very successful model has been the source of a great

deal of attention in recent years (see [14] and [20] for background on data stream computation), however the techniques developed have largely been randomised whereas our interest is in deterministic solutions.

Our main contribution is a black box for converting offline approximate matching algorithms into efficient online ones. That is, it ensures that the approximate matching algorithm accomplishes its task for the  $i$ th input character without requiring the  $i + 1$ th. The method is deterministic and bounds the worst case running time *per input character* as well as ensuring that the overall running time is within a log factor of the best known offline algorithm. It is an important feature of our method that its running time is not amortised. This is because when processing streaming data it may not be realistic to wait for long periods of time between individual input characters.

Our technique can be applied to a wide class of approximate matching algorithms overcoming one of the main restrictions on their use in data streaming applications. A particularly useful subset that we focus on in this paper includes problems whose distance function  $\Delta$  is defined so that  $\Delta(x, y) = \sum_{j=1}^m \Delta(x_j, y_j)$ , for strings  $x$  and  $y$  and  $|x| = |y| = m$ . In other words, distance functions between two strings where the distance is simply measured as the sum of the distances between individual symbols. Many of the most common and widely studied approximate matching problems fall into this category including exact matching with wildcards, matching under the Hamming norm,  $k$ -mismatch and matching under the  $L_2$  and  $L_1$  norms. As a result, we provide fast deterministic online algorithms for each one of these problems.

The overall structure of the paper is as follows. In Section 2 we summarise the main results of the paper. In Section 3 we present the main black box solution and in Section 4 we discuss space lower bounds. Finally we conclude with some open problems in Section 5.

## 2 Our Results

Our black box approach converts an offline approximate matching algorithm into efficient online algorithm. Let  $T(n, m)$  be the total running time of the best known offline approximate matching algorithm for the problem being considered. The main results we present are as follows:

- We show how offline approximate matching algorithms can be turned into online algorithms with strict bounds on the computation time per input character. The main idea is to split the pattern into  $O(\log m)$  subpatterns of successively halving length and to perform searches in parallel on carefully chosen partitions of the text. The partitions are chosen so that the work needed to compute the distance to a sliding window of the text is started  $O(m)$  characters before it is needed.

Specifically, for each subpattern of length  $m'$ , we start the approximate matching algorithm  $m'/2$  characters before its result is required. This work is carried out in parallel (by time slicing for example) with searches involving a subset of the remaining subpatterns. An auxiliary array of size  $O(m)$

is sufficient to keep track of the cumulative counts of the distances found so far. The online algorithm takes  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1})/n)$  time per input character. As  $\sum_{j=1}^{\log_2 m} T(n, 2^{j-1}) \leq T(n, m) \log(m)$ , this gives a near optimal deterministic solution to a wide class of online approximate matching problems.

- A small adjustment to the algorithm allows us to report the distance to a sliding window in the text in constant time after a new character arrives. Although the computation time per character is unchanged, we are able to move the majority of the work for future symbols until after a new symbol has been processed. This provides a solution for online approximate matching in a model where instant answers are needed once new data arrives.
- Applications of our black box method to exact matching with wildcards, matching under the Hamming norm, approximating the Hamming norm,  $k$ -mismatch and matching under the  $L_2$  and  $L_1$  norms result in algorithms that take  $O(\log^2 m)$ ,  $O(\sqrt{m \log m})$ ,  $O(\log^2 m/\epsilon^2)$ ,  $O(\sqrt{k \log k} \log m)$ ,  $O(\log^2 m)$  and  $O(\sqrt{m \log m})$  time per character respectively.
- Finally we argue that the space requirements for the online approximate matching problem are optimal in the deterministic setting under the assumption that the offline matching algorithm requires  $O(m)$  space. This follows immediately from an  $\Omega(m)$  communication complexity lower bound for computing “sum-type” functions between strings.

### 3 The Black Box for Online Approximate Matching

The black box we present will make repeated calls to an offline approximate pattern matching algorithm which we call *offline-pm*. In order to simplify some of the explanation, we assume that the running time  $T(n, m)$  of *offline-pm* can be expressed as  $nT'(m)$  with  $T'(m) \in O(m)$ . This assumption is reasonable as the types of pattern matching problem we consider can all be solved naively in  $O(nm)$  time. As a result of this simplification we have  $O(n/mT'(cm, m)) = O(T(n, m))$  for constant  $c > 1$ . We will also at times refer to a call to *offline-pm* as a search for the sake of brevity.

The basic idea of our black box is to split the pattern into  $O(\log m)$  consecutive subpatterns each having half the length of the previous one. In this way  $P_1 = p[1, \dots, m/2]$  and subpattern  $P_j$  has length  $m2^{-j}$  for  $1 \leq j \leq \log_2(m)$ .  $P_{\log_2(m)+1}$  is set to be the last character of the pattern. We then run *offline-pm* for each subpattern against the whole of the text. The distances found can then be added to an auxiliary array  $C$ . Specifically, for any subpattern starting at position  $j$  of the pattern, its distance to a substring starting at position  $i$  of the text will be added to the count at  $C[i - j + 1]$ . At the end of this step  $C$  will contain  $\Delta(p, t[i, \dots, i + m - 1])$  for every location  $i$  in  $t$ .

This algorithm will call *offline-pm*  $O(\log m)$  times and requires  $O(n)$  extra space for the auxiliary array. The space requirement can be reduced to  $O(m)$  by partitioning the text. For any subpattern of length  $m'$ , we partition the text into  $n/(m' - 1)$  overlapping substrings of length  $2m'$ , each with an overlap of

length  $m'$  with the previous partition. If we run offline-pm on each partition separately, the total time complexity over the whole text for each subpattern is  $O((n/m)T(2m, m)) = O(T(n, m))$ . The distances for each subpattern can be added to the auxiliary  $C$  in the same way as before. However, now we only need store one auxiliary array of size  $m$  at most.

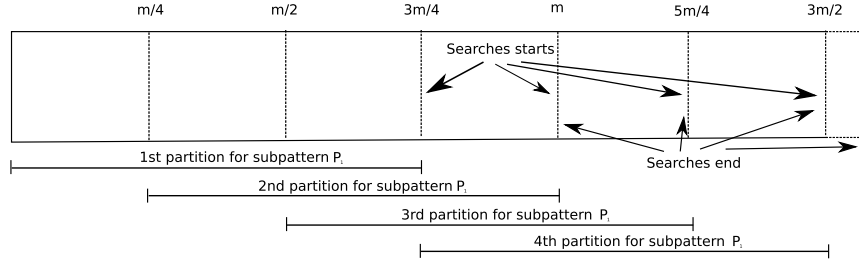
This space reduced algorithm can easily be made online by a lazy execution of the searches performed. For each subpattern  $P_j$  there is a list of locations associated with it which marks out the start and end of its associated partitions of the text. These locations do not have to be stored explicitly as they are easily computable as they are needed. For example, for  $P_1 = p[1, \dots, m/2]$ , the first partition is  $t[1, \dots, m]$ , the second is  $t[m/2, 3m/2]$  and so on. When the  $i$ th character is read in from the text, offline-pm can now perform the searches for all the subpatterns which have a partition finishing at position  $i$ . This online algorithm runs in  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1}))$  time and  $O(m)$  extra space. However the solution is not yet satisfactory as we might potentially have to wait for more than  $T(m, m/2)$  time *after* a new character arrives before we are able to compute the distance to the new sliding window. As an example, when the  $m$ th character of  $t$  is read in, the first search involving  $P_1$  commences for the partition  $t[1, \dots, m]$ , thereby delaying the computation of  $\Delta(p, t[1, \dots, m])$  unacceptably. Our aim is to ensure that the maximum computation time per character is limited to  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1})/n)$ .

### Bounding the Maximum Time per Character

We can bound the maximum time per character by performing more work earlier on. Instead of splitting the text into partitions of size  $2m'$  per subpattern, we change the partition size to  $3m'/2$ . The overlap between partitions is maintained at  $m'$  to ensure no matches are missed.

If the  $i$ th character of the text is read in, searches will now be performed for all subpatterns which have a partition of the text ending at position  $i$ . The result of a search involving a subpattern will now not be needed until the  $i + m'/2$ th character is read in. That is  $m'/2$  characters after the relevant search has been performed. In this way, whenever a new character is read in, the results for all the subpatterns needed to compute the distance to the new sliding window are already known, except for the last character of the pattern. This last comparison can be carried out in constant time. Figure 1 shows the partitioning of the text for the first subpattern  $P_1$  and the start and finish times of the searches performed.

In order to guarantee the desired upper bound for the amount of work carried out per character, we need to spread the work out evenly over the time it takes to input the text. For each subpattern, the work for a particular search does not have to be completed until  $m'/2$  characters after it starts and so we can set this work to be performed over the period between reading in the  $i$ th and the  $i + m'/2$ th character. As an example, the first search for  $P_1$  starts when  $t[3m/4]$  is read in and completes at the time that  $t[m]$  is.  $P_2$  will on the other hand start to calculate its contribution to  $\Delta(p, t[1, \dots, m])$  when  $t[7m/8]$  is read in. The result from these searches and from all the other subpatterns, will be scheduled



**Fig. 1.** Partitioning of the text for subpattern  $P_1 = p[1, \dots, m/2]$

to complete by the time  $t[m]$  arrives. As there will be a number of searches scheduled to work in this period we will need to perform the work in parallel by time slicing. To guarantee the bound on the work per character without requiring any internal knowledge of offline-pm, we only require an upper bound for the running time of offline-pm for a subpattern of length  $m'$  which we then divide by  $m'/2$ . This gives the amount of work to carry out per input character.

Algorithm 1 gives an overview of the whole process.

**Input:** Pattern  $p$ , a the streaming text  $t$  and offline-pm  
**Output:**  $\Delta(p, t[i - m + 1, \dots, i])$  for each  $i \geq m$  of streaming text  
 Initialisation;  
     Split  $p$  into  $\log_2 m$  subpatterns  $P_j$  of length  $m/2^j$ , for  $1 \leq j \leq \log_2 m$ ;  
     For each  $P_j$ , calculate its partition start and end points;  
**foreach** symbol  $t[i]$  read in **do**  
     Add  $\Delta(p[m], t[i])$  to  $C[i - m + 1]$ ;  
     Wait for results of offline-pm searches due to end at position  $i$ ;  
     Output  $C[i - m + 1]$ ;  
     Start offline-pm searches for each subpattern  $P_j$  which has a partition ending at  $i$ ;  
**end**

**Algorithm 1.** Black box algorithm for online pattern matching

By making a small adjustment to the scheduling of the work we can also guarantee that all but a constant amount of the work needed to compute  $\Delta(p, t[i - m + 1, \dots, i])$  will have been completed *before* the  $i$ th character is read in. Although no change is made to the total work per character, the ability to control at which point work is carried out can have applications when data arrives in bursts. For example, if there is some pause in the data stream before new characters arrive. The following Theorem summarises the main result.

**Theorem 1.** *Algorithm 1 solves the online approximate problem in  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1})/n)$  time per input character and  $O(m)$  space. Further, with a small*

modification it can report the distance to a new sliding window in constant time after a text symbol arrives.

*Proof.* The total time taken by Algorithm 1 is  $O(T(n, m'))$  per subpattern of length  $m'$  making a total of  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1}))$  time overall for all subpatterns. The work performed by the calls to offline-pm is evenly spread over the whole length of the text. Therefore, the total amount of work per character is  $O(\sum_{j=1}^{\log_2 m} T(n, 2^{j-1})/n)$ .

The space required for the auxiliary array is  $O(m)$ . We also have to consider the space overhead of offline-pm as there can be  $O(\log m)$  searches running simultaneously. Under the assumption that each individual search requires  $O(m)$  space, the total space requirement is less than  $\sum_{j=1}^{\log_2 m} (c2^{j-1}) = O(m)$  overall.

In order to output  $\Delta(p, t[i - m + 1, \dots, i])$  in constant time after  $t[i]$  is read in we need only ensure that a search is completed  $m'/2 - 1$  (rather than  $m'/2$ ) characters after the end of a partition. In this way, when the  $i$ th character is read in only  $\Delta(p[m], t[i])$  will remain to be computed. This modification does not affect the time complexity of the algorithm overall.  $\square$

We can now apply our black box to a number of well known matching problems, giving the following time complexities per input character.

**Corollary 1.** *Algorithm 1 applied to the fastest known offline pattern matching algorithms for the Hamming norm,  $k$ -mismatch and matching under the  $L_2$  and  $L_1$  norms gives online algorithms that take  $O(\log^2 m)$ ,  $O(\sqrt{m \log m})$ ,  $O(\sqrt{k \log k \log m})$ ,  $O(\log^2 m)$  and  $O(\sqrt{m \log m})$  time per character respectively.*

## 4 Space Lower Bound for Deterministic Online Approximate Matching

It would seem desirable to reduce the space requirements even further in order to increase the practicality of processing data streams. Unfortunately, there is an  $\Omega(m)$  communication complexity lower bound for “sum-type” functions [21] which covers the additive distance functions we have been most interested in. The space lower bound for any deterministic approximate pattern matching algorithm follows directly from the communication complexity lower bound by a standard argument that we briefly summarise. Assuming the communication complexity lower bound, the proof of the space lower bound is by contradiction. If Alice can preprocess the pattern to use  $o(m)$  space and then starts her online pattern matching algorithm, she could then transfer a snapshot of the current state to Bob who could then carry on running the algorithm on his string. Bob would then find the distance to his string having received only  $o(m)$  items of data, thereby giving the desired contradiction.

## 5 Discussion

The method we have developed is applicable to a wide class of previously offline approximate matching algorithms. By choosing a black box approach we have

not investigated whether particular pattern matching algorithms might be more easily converted to efficient online algorithms without any extra time cost. Also, although we have shown that the space required by our approach is optimal for a wide range of problems, an interesting question is whether randomisation can allow us to solve the same problems with only  $o(m)$  space as the communication complexity bounds will no longer hold.

## Acknowledgements

The authors would like to thank Inbok Lee and Ashley Montanaro for their helpful comments on a draft of this paper.

## References

- [1] Abrahamson, K.: Generalized string matching. *SIAM journal on Computing* 16(6), 1039–1051 (1987)
- [2] Amir, A., Farach, M.: Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation* 118(1), 1–11 (1995)
- [3] Amir, A., Lipsky, O., Porat, E., Umanski, J.: Approximate matching in the  $L_1$  metric. In: Apostolico, A., Crochemore, M., Park, K. (eds.) *CPM 2005*. LNCS, vol. 3537, pp. 91–103. Springer, Heidelberg (2005)
- [4] Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with  $k$  mismatches. *J. Algorithms* 50(2), 257–275 (2004)
- [5] Atallah, M.J.: Faster image template matching in the sum of the absolute value of differences measure. *IEEE Transactions on Image Processing* 10(4), 659–663 (2001)
- [6] Clifford, P., Clifford, R.: Simple deterministic wildcard matching. *Information Processing Letters* 101(2), 53–54 (2007)
- [7] Clifford, P., Clifford, R., Iliopoulos, C.S.: Faster algorithms for  $\delta, \gamma$ -matching and related problems. In: Apostolico, A., Crochemore, M., Park, K. (eds.) *CPM 2005*. LNCS, vol. 3537, pp. 68–78. Springer, Heidelberg (2005)
- [8] Clifford, R., Iliopoulos, C.: String algorithms in music analysis. *Soft Computing* 8(9), 597–603 (2004)
- [9] Clifford, R., Efremenko, K., Porat, E., Rothschild, A.:  $k$ -mismatch with don't cares. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) *ESA 2007*. LNCS, vol. 4698, pp. 151–162. Springer, Heidelberg (2007)
- [10] Clifford, R., Porat, E.: A filtering algorithm for  $k$ -mismatch with don't cares. In: Ziviani, N., Baeza-Yates, R. (eds.) *SPIRE 2007*. LNCS, vol. 4726, pp. 130–136. Springer, Heidelberg (2007)
- [11] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms*. MIT Press, Cambridge (1990)
- [12] Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: *Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 592–601 (2002)
- [13] Fischer, M., Paterson, M.: String matching and other products. In: Karp, R. (ed.) *Proceedings of the 7th SIAM-AMS Complexity of Computation*, pp. 113–125 (1974)

- [14] Henzinger, M.R., Raghavan, P., Rajagopalan, S.: Computing on data streams. In: External memory algorithms, pp. 107–118. American Mathematical Society, Boston (1999)
- [15] Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 166–173 (1998)
- [16] Kalai, A.: Efficient pattern-matching with don't cares. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 655–656 (2002)
- [17] Kosaraju, S.R.: Efficient string matching (1987) (manuscript)
- [18] Landau, G.M., Vishkin, U.: Efficient string matching with  $k$  mismatches. Theoretical Computer Science 43, 239–249 (1986)
- [19] Lipsky, O., Porat, E.: Approximate matching in the  $L_\infty$  metric. In: String Processing and Information Retrieval, 12th International Symposium (SPIRE 2005). LNCS, pp. 331–334. Springer, Heidelberg (2005)
- [20] Muthukrishnan, S.: Data streams: algorithms and applications. In: SODA 2003: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, p. 413 (2003)
- [21] Tamm, U.: Communication complexity of sum-type functions invariant under translation. Inf. Comput. 116(2), 162–173 (1995)