Looking for MUM and DAD: text–text comparisons do help ⋆

Mike Paterson,[1] Shlomit Tassa,[2] Uri Zwick[2]

[1] Department of Computer Science, University of Warwick, UK,
[2] Department of Computer Science, Tel Aviv University, Israel

Abstract. It is known that about $\frac{4n}{3}$ comparisons are needed, in the worst case, to find all the occurrences of the string `aba` in a text of length n if only pattern–text comparisons are allowed. We show that if text–text comparisons can be used then only about $\frac{5n}{4}$ comparisons are required. This gives the first example in which text–text comparisons provably help.

1 Introduction

We consider one of the simplest possible string matching problems: what is the number of comparisons needed, in the worst case, to find all the occurrences of the pattern `aba` in a text of length n. Surprisingly, the exact answer to this question is not yet known.

There has been much recent research analysing the exact complexity of algorithms on strings. Particular attention has been given to comparison-based string matching algorithms. The only operations such algorithms are allowed to perform on the pattern and text characters are pairwise comparisons. To each such comparison the algorithm gets a 'yes' (the characters compared are equal) or a 'no' (the characters compared are not equal) answer.

The well-known Knuth–Morris–Pratt (KMP) algorithm [13] is a comparison-based algorithm. It uses only pattern–text comparisons and performs at most $2n$ comparisons in order to find all the occurrences of a pattern of length m in a text of length n. In contrast, the also well-known Boyer–Moore (BM) algorithm [3] is not comparison-based as it uses the pattern and text characters as indices to tables. A comparison-based variant of the BM algorithm was shown by Cole [4] to perform at most $3n + o(n)$ comparisons.

Apostolico and Crochemore [1] and Colussi [5] designed variants of the KMP algorithm requiring at most $\frac{3n}{2}$ comparisons. Galil and Giancarlo [11] improved their results and obtained an algorithm that performs at most $\frac{4n}{3}$ comparisons. All these algorithm use only pattern–text comparisons. Furthermore, they are *on-line* algorithms, i.e., they only access the text through a sliding window whose size is equal to the length of the pattern.

⋆ Supported in part by the ESPRIT Basic Research Action Programme of the EC under contract No. 7141 (project ALCOM II).

Breslauer and Galil [2] and Cole and Hariharan [6] showed that the string matching problem actually becomes easier as the length of the pattern increases. Breslauer and Galil [2] developed an algorithm that performs at most $(1+O(\frac{\log m}{m}))n$ character comparisons for texts of length n and patterns of length m. Cole and Hariharan [6] improved this by obtaining an algorithm that performs at most $(1+\frac{8}{3(m+1)})n$ comparisons for patterns of length m. It is therefore interesting to study the complexity of finding the occurrences of particular short patterns. In particular, `aba` (or MUM or DAD) is probably the hardest pattern to find.

Galil and Giancarlo [10] and Cole *et al.* [7] proved lower bounds on the number of comparisons that comparison-based string matching algorithms must perform in the worst case. In particular, Galil and Giancarlo showed that any on-line algorithm must perform, in the worst case, at least $\frac{4n}{3}$ comparisons to find all occurrences of `aba` in a text of length n. Cole *et al.* showed that $\frac{4n}{3}$ comparisons are also necessary for *off-line* algorithms, i.e., algorithms that have general random access to the text and pattern, provided that only pattern–text comparisons are allowed. These two lower bounds are easily seen to be tight.

In this work we show that the $\frac{4n}{3}$ lower bound fails to hold if text–text comparisons are allowed. We describe an off-line algorithm that finds all occurrences of `aba` in a text of length n using at most $\frac{5n}{4}$ comparisons. This exhibits the first example in which text–text comparisons provably help. However, the best lower bound known is that of $\frac{6n}{5}$ obtained by Cole *et al.* [7], so the exact complexity of the problem is not yet resolved.

2 Notation and Preliminaries

We consider algorithms which access the text t and the pattern w only through queries of the form "$w[i]=t[j]$?" or "$t[i]=t[j]$?", and measure the complexity in terms of the number of such comparisons required in the worst case.

It can be shown that for any string w there is a constant $c(w)$ such that the worst-case comparison complexity of finding all occurrences of the pattern w in a text of length n is $c(w)n+o(n)$. We define also the constant $c^*(w)$ corresponding to algorithms which can make only pattern–text comparisons, i.e., those of the form "$w[i]=t[j]$?".

These constants are for unrestricted *off-line* algorithms which have random access to all the characters of the text. We will also consider *on-line* algorithms, where access to the text is only through a sliding window of size $|w|$. The window is moved from left to right and the algorithm is only allowed to slide the window past a text position when it has already reported whether or not an occurrence of the pattern starts at that text position. The corresponding constants for on-line algorithms are denoted by $c_0(w)$ and $c_0^*(w)$ respectively. Algorithms using larger windows are also of interest. We call these *window* or *finite look-ahead* algorithms. If the window is of size $|w|+k$, the corresponding two constants will be denoted by $c_k(w)$ and $c_k^*(w)$.

Suppose $|w| = m$. We say that z $(1 \leq z \leq m)$ is *a period* of w if and only if $w[i] = w[i+z]$ for every $1 \leq i \leq m - z$. Let z_1 be the minimal period of w. (A minimal period exists since m is always a period of w.) Let z_2 be the minimal period of w which is not divisible by z_1. If such a second period does not exist we set $z_2 = \infty$. We call z_1 *the period* of w and z_2 *the second period* of w.

We consider text strings over the (finite or infinite) alphabet Σ which contains the symbols `a`, `b`, and at least one other symbol. In this paper we make extensive use of bounds proved in [7], but focus our attention on patterns of the form $\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$, and even more on the single pattern `aba` in particular (as suggested by the title of the paper). Determining the complexity of looking for `aba` already presents difficult combinatorial problems. It also seems likely that `aba` is the hardest pattern to find!

3 Lower Bounds

The following lower bounds are proved in [7].

Proposition 1. *If w is a string and z_1, z_2 are its first and second periods then:*

1. $c(w) \geq 1 + \frac{1}{z_1+z_2}$;
2. *if* $2z_2 - z_1 \leq |w|$ *then* $c(w) \geq 1 + \frac{1}{z_2}$;
3. $c_0(w) \geq 1 + \frac{1}{z_2}$.

Fact. For $w = \mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$, where $k, \ell \geq 1$, we have $z_1 = \max\{k, \ell\} + 1$ and $z_2 = \max\{k, \ell\} + 2$.

Corollary 2. *If $p = \max\{k, \ell\}$ and $q = \min\{k, \ell\} \geq 1$ then:*

1. $c(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) \geq 1 + \frac{1}{2p+3}$;
2. *if* $q \geq 2$ *then* $c(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) \geq 1 + \frac{1}{p+2}$;
3. $c_0(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) \geq 1 + \frac{1}{p+2}$.

For small look-ahead we have a new generalization of Proposition 1.

Theorem 3. *If z_1 and z_2 are the first and second periods of w, and $r = \min\{z_1, z_2 - z_1\}$, then $c_r(w) \geq 1 + \frac{1}{z_1+z_2-r} = 1 + \frac{1}{\max\{2z_1, z_2\}}$.*

Proof. The Adversary will construct stage-by-stage a text string which is covered by (usually overlapping) occurrences of w. In such cases the algorithm requires at least n 'yes' responses for a text of length n (see Lemma 3.1 [7]). We will show that, even if an algorithm can take advantage of the restricted strategy employed by the Adversary, it will still be forced to get a 'no' response for each shift of the window by at most $\max\{2z_1, z_2\}$ steps.

Suppose that the algorithm has just found an occurrence of w and has then been able to move its window z_1 places right. The Adversary will now guarantee that the next part of the text consists of either (a) two overlapping occurrences of w displaced by z_1 and $2z_1$ to the right relative to the found occurrence, or else (b) an occurrence of w displaced by distance z_2 to the right. Since the window size is only $|w| + \min\{z_1, z_2 - z_1\}$, the algorithm can access nothing to the right of these alternative continuations until it has decided whether or not case (a) holds.

As long as the algorithm asks questions whose answer is the same for each of these two continuations, this common answer will be given. At some stage the algorithm must ask a question that differentiates between the two continuations. The answer to this question will be 'no'. The algorithm may now take advantage of the guarantee from the Adversary and verify the continuation (a) or (b). As a result, the algorithm finds two or one occurrences of w respectively and achieves a shift of at most $\max\{2z_1, z_2\}$ for the price of one 'no' response. □

Corollary 4. *If* $w = \mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$, $p = \max\{k, \ell\}$ *and* $q = \min\{k, \ell\} \geq 1$ *then* $c_1(w) \geq 1 + \frac{1}{2p+2}$.

4 Previous Upper Bounds for $\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$

To represent the state of knowledge about a text string during the course of a window algorithm we shall use the *information string* notation introduced in [7]. The text in the window is represented by a string over the alphabet $\{0, \mathtt{a}, \mathtt{b}, \mathtt{A}, \mathtt{B}\}$, where a 0 indicates that no information is recorded about that position, `a` or `b` indicates that the character is known to be `a` or `b` respectively, while `A` or `B` means that the character is known *not* to be an `a` or `b` respectively. Further information may be added in other ways. A pattern–text query will be called an `a`-*query* or a `b`-*query* according to whether the $w[i]$ used is `a` or `b`. We sometimes point out the next position to be queried in an algorithm by underlining that position in the information string: a single underline for an `a`-query, a double underline for a `b`-query.

A very simple on-line algorithm for finding occurrences of $\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$ was given in [7]. It makes a sequence of '`a`?' and '`b`?' queries. In the case of a mismatch or if an occurrence of the pattern has been verified, the window is shifted to the next position at which a pattern occurrence is possible. The information string representing the state before each step has the form uxv, where $u \in \{0, \mathtt{a}\}^k$, $v \in \{0, \mathtt{a}\}^\ell$, and $x \in \{0, \mathtt{A}, \mathtt{b}\}$. The state can be written in the specified form because, after any necessary window shift, the information string must be consistent with the pattern, and we *choose to forget* any negative information represented by '`B`'. The $(k+1)$-st position in the window is called the `b`-*position* and all the others `a`-*positions*. An `a`-position is always queried for an `a`. A `b`-position is always queried for a `b`. The algorithm is described below.

```
REPEAT
    IF there is some 0 in the information string
        THEN query the rightmost 0
    ELSE {x = A} query the b-position
UNTIL text string exhausted.
```

In [7] we analyse the worst-case complexity of this on-line algorithm and derive the following result.

Proposition 5. *For every $k, \ell \geq 1$ we have $c_0^*(\mathtt{a}^k\mathtt{b}\mathtt{a}^\ell) \leq 1 + \frac{1}{\max\{k,\ell\}+2}$.*

With Corollary 2(2), this yields tight bounds, provided $\min\{k, \ell\} \geq 2$.

Corollary 6. *If $w = \mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$, $p = \max\{k, \ell\}$, and $\min\{k, \ell\} \geq 2$ then $c(w) = c^*(w) = c_r(w) = c_r^*(w) = 1 + \frac{1}{p+2}$ for every $r \geq 0$.*

Thus, when $\min\{k, \ell\} \geq 2$, an on-line algorithm using only pattern–text comparisons performs as well as any arbitrary off-line algorithm. However, when $\min\{k, \ell\} = 1$, a potential gap emerges. From the above proposition and Corollary 2, we have the following corollary.

Corollary 7. *If $w = \mathtt{a}^k\mathtt{b}\mathtt{a}^\ell$, $p = \max\{k, \ell\}$, and $\min\{k, \ell\} = 1$ then $c_0(w) = c_0^*(w) = 1 + \frac{1}{p+2} \geq c(w) \geq 1 + \frac{1}{2p+3}$.*

For the pattern abaa, we were able to improve on the above algorithm by using a window of size 8. This shows that look-ahead is indeed helpful.

Proposition 8 [7]. $c_0^*(\mathtt{abaa}) = c_0(\mathtt{abaa}) = \frac{5}{4} > c_4^*(\mathtt{abaa}) = c_4(\mathtt{abaa}) = c(\mathtt{abaa}) = \frac{8}{7}$.

5 Looking for aba

In the remainder of the paper we focus entirely on the pattern string aba. We shall see that certain problems or sets of problems are *equivalent*, in the sense that any optimal off-line algorithm for the one can be trivially transformed into an algorithm for the other, which has an isomorphic decision tree and so uses the same numbers of comparisons in corresponding cases. We note without proof some obvious equivalences arising from the special nature of the pattern aba.

Lemma 9. *Let α and β be arbitrary strings over the information alphabet $\{0, \mathtt{a}, \mathtt{b}, \mathtt{A}, \mathtt{B}\}$.*

(i) The problem $\mathtt{A}\alpha$ is equivalent to the problem α.
(ii) The problem $\mathtt{B}\alpha$ is equivalent to the problem 0α.
(iii) The problem α is equivalent to the problem α^R, where α^R denotes the reversal of the string α.
(iv) The problem $\alpha\mathtt{a}\beta$ is equivalent to the pair of problems $\alpha\mathtt{a}$ and $\mathtt{a}\beta$.
(v) The problem $\alpha\mathtt{AA}\beta$ is equivalent to the pair of problems α and β.
(vi) The problem $\alpha\mathtt{BB}\beta$ is equivalent to the pair of problems $\alpha 0$ and 0β.

Thus for example the problem a00AA00a000a is equivalent to the set of three problems {a000a, a00, a00}.

Using such decompositions, we derived a lower bound of 4/3 for $c^*(\mathtt{aba})$.

Proposition 10 [7]. $c_0(\mathtt{aba}) = c_0^*(\mathtt{aba}) = c^*(\mathtt{aba}) = \frac{4}{3} \geq c(\mathtt{aba}) \geq \frac{6}{5}$.

Our main results in the next two sections will show that, for the pattern aba, text–text comparisons do indeed help.

6 A Finite Configuration

We show in Figure 1 the number of comparisons which are sufficient and necessary in the worst case to solve some small problems when *only pattern–text comparisons* are used. These numbers can be determined by careful analysis considering at each stage the set of possible remaining places for occurrences of `aba`. In some cases we have indicated one possible next comparison in an optimal algorithm.

configurationt	comparisons	configuration	comparisons
$\mathtt{a00}$	2	$\mathtt{a0Ba}$	2
$\mathtt{a\underline{0}00}$	3	$\mathtt{a0Aa}$	2
$\mathtt{a0\underline{0}00}$	4	$\mathtt{a00Aa}$	3
$\mathtt{a0a}$	1	$\mathtt{a0\underline{0}A00a}$	5
$\mathtt{a00a}$	3	$\mathtt{a00\underline{B}00a}$	6
$\mathtt{a000a}$	4	$\mathtt{a00\underline{0}00a}$	7

Fig. 1. Small configurations and their individual costs using pattern–text comparisons.

In addition, it is not difficult to show by induction the following result for a simple family of texts.

Theorem 11. *Let $T(n)$ be the cost of finding* `aba`*'s in the configuration* $\mathtt{a0}^n\mathtt{a}$ *using pattern–text comparisons only. Then*

$$T(n) = \left\lfloor \frac{4n+1}{3} \right\rfloor .$$

When text–text comparisons are allowed the difficulty of analysing small configurations increases. We used a computer to extend our search and soon discovered, to our surprise, that only six comparisons are now needed for the text string `a00000a`; an algorithm is shown in Figure 2. The first step is a comparison between the second and fourth positions of the text string. The left and right branches of the decision tree show the continuation according to whether the result is positive or negative respectively. On the left the next step is an `a`-query on the second text position; on the right, a `b`-query on the same position. The leaf configurations show the remaining queries to be made. For the leftmost leaf, we see from Figure 1 that three comparisons suffice for `a00a`. At the third leaf, the notation `C` means "not `a` and not `b`", so the leaf configuration is equivalent to `00a` (or `a00`).

From Figure 1 it can be seen that seven comparisons are needed for the text `a00000a` with the pattern `aba` in the worst case when text–text comparisons are forbidden. This gives the smallest concrete situation that we know in which text–text comparisons do help.

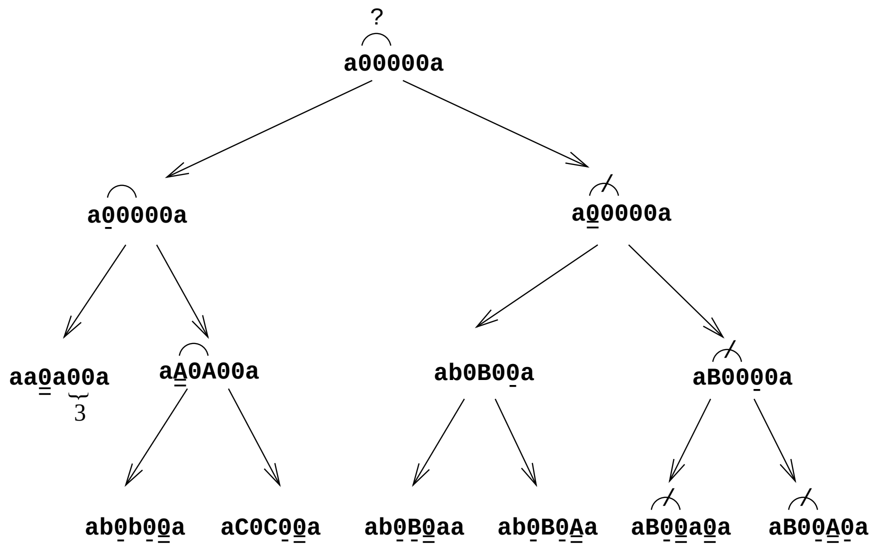

Fig. 2. Algorithm for the text string a00000a.

7 Showing that c(aba) < c^*(aba)

Having seen the computer-generated algorithm for a00000a, we were able to take advantage of the ingenious first (text–text) comparison used there to design a more efficient general aba-finding algorithm.

Theorem 12. *c(aba) $\leq c_1$(aba) $= \frac{5}{4} < c^*$(aba).*

Proof. The lower bound, c_1(aba) $\geq \frac{5}{4}$, is given by Corollary 4. The upper bound is established by the algorithm presented in Figure 3 in the form of a decision tree. The configurations are information strings for an algorithm with a window of size four. Each leaf of the tree is one of the configurations 0000 or a000, and represents a return to the root of the tree or its left successor. After the result of each query, the window is shifted to the right where possible. A label x:y on an arc indicates that x (obvious) queries are made and the window is shifted right by y places. (An unlabelled arc corresponds to the default 1:0.) For example, consider the root query $\mathtt{00\underline{0}0}$. If the result is positive the information string is 00a0, and after the two queries $\mathtt{\underline{0}\,\underline{\underline{0}}a0}$ the window can be shifted two places to the right.

In the diagram the arcs corresponding to the worst-case executions of the algorithm are shown as heavy lines. For executions following these arcs the average number of window shifts per query is 4/5, which establishes the claimed upper bound: c_1(aba) $\leq 5/4$. □

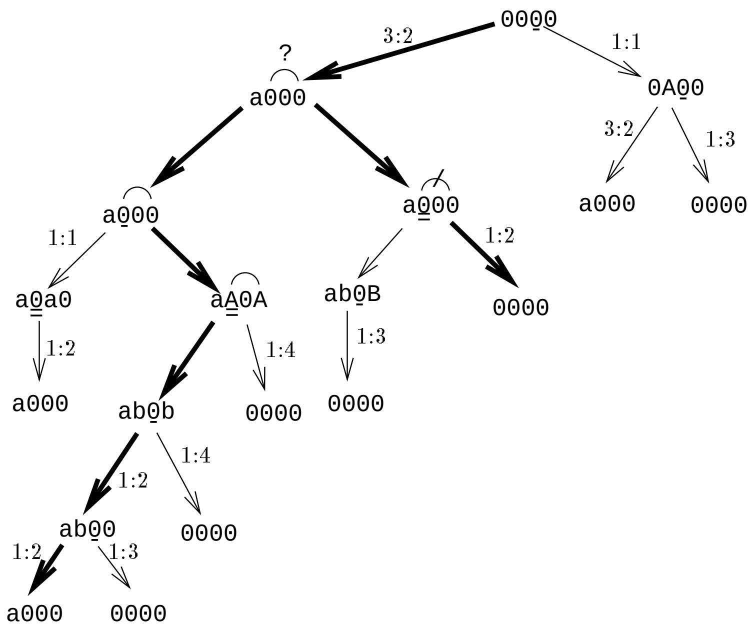

Fig. 3. Algorithm showing $c(\texttt{aba}) \leq 5/4$.

8 Further Results

As we have seen in the two previous sections, computer assistance has been valuable in searching for efficient pattern-matching algorithms. We have investigated the complexity of some short text strings as well as window algorithms with small window size. In this section we report on some of these results. Work is still continuing.

For finding `aba`'s in fixed text strings, we can try to establish the complexity by exploring the full tree of possible algorithms, taking advantage of reductions, decompositions and equivalences where possible.

We would like to know whether `a00000a` is the smallest problem for which text–text comparisons are helpful. An interesting extension of this kind of question is to consider *sets of* configurations. When text–text comparisons are permitted, it may no longer be true that the query complexity of a set of configurations is equal to the sum of the complexities of the individual configurations. There could be the possibility of economies in *mass production*, i.e., solving ℓ disjoint pattern matching problems might sometimes require fewer than ℓ times the number of comparisons needed to solve each of these problems separately. (A similar phenomenon has been usefully employed in the context of selection algorithms over ordered sets. Mass production of partial orders was used by Schönhage, Paterson and Pippenger [14], and recently by Dor and Zwick [8], to reduce the number of comparisons needed to find the median.) One negative result so far is that mass-production does not help for $\mathtt{a00a}$.

For pattern-matching on arbitrary texts, if we restrict attention to algorithms with a fixed window size then the number of possible configurations (information strings) is finite (though it grows rapidly with window size). We can construct by computer the graph representing all possible queries for an algorithm at each stage and both possible responses to the query. The situation can be regarded as a two-person game, played between the Designer of algorithms and the Adversary, who gives responses to force worst-case behaviour. The players move alternately on the resulting bipartite graph: for each configuration node, the Designer selects a next query (represented by an arc to a query node); for each such query node, the Adversary selects one of the two possible outcomes, choosing an arc to a configuration node. The measure of concern is the average number of queries per window shift along the resulting path. The Designer is trying to minimise this average, while the Adversary tries to maximise. The constant $c(w)$ is both the smallest average which can be ensured by the Designer and the largest average which can be ensured by the Adversary.

This is a formulation of the problem as a *mean payoff game*. These were introduced by Ehrenfeucht and Mycielski [9] and independently considered by Gurvich, Karzanov and Khachiyan [12]. Zwick and Paterson [15] have recently given a polynomial-time algorithm for finding optimal strategies for the kind of mean payoff games arising from pattern-matching (though only pseudo-polynomial for the most general formulation). Using this kind of algorithm we have obtained the following lower bound on algorithms for $\mathtt{aba}$-finding with a window of size 5.

Theorem 13. $c_2(\mathtt{aba}) \geq \frac{5}{4}$.

Corollary 14. $c_0(\mathtt{aba}) = \frac{4}{3} > c_1(\mathtt{aba}) = c_2(\mathtt{aba}) = \frac{5}{4} \geq c(\mathtt{aba}) \geq \frac{6}{5}$.

9 Conclusion and Open Problems

We have reviewed some recent results on comparison-based algorithms for pattern matching. In this paper we focussed particularly on the single pattern string $\mathtt{aba}$, which already offers challenging problems. This string also seems to the hardest pattern to find, though this is not yet proven. An unexpected outcome of our research is that comparisons between pairs of positions in the text (rather than just those between pattern and text) are required for optimal algorithms.

We know that text–text comparisons help for finding $\mathtt{aba}$ in the text string $\mathtt{a00000a}$, but do not know if this is the simplest such example. A more fundamental open problem is whether text–text comparisons give rise to economies using mass-production (see Section 8).

An interesting structural question concerning window algorithms is whether there is an infinite hierarchy with window size for any fixed pattern. Is there some pattern w, perhaps even $\mathtt{aba}$, such that for every k there exists a k' for which $c_k(w) < c_{k'}(w) < c(w)$?

For any w, $c_k(w)$ and $c_k^*(w)$ must be rational numbers for all k, but we do not know whether this is also true for $c(w)$ and $c^*(w)$.

Is it always the case that $c_0(w) = c_0(w^R)$ and $c_0^*(w) = c_0^*(w^R)$, where w^R is the reverse of w? Clearly this holds for c and c^*.

The most immediate open problems are to determine the exact value of $c(\texttt{aba})$ and to see whether `aba` is indeed the hardest pattern to find.

References

1. A. Apostolico and M. Crochemore, *Optimal canonization of all substrings of a string,* Information and Computation, 95 (1991), pp. 76–95.
2. D. Breslauer and Z. Galil, *Efficient comparison based string matching,* Journal of Complexity, 9 (1993), pp. 339–365.
3. R. Boyer and S. Moore, *A fast string matching algorithm,* CACM, 20 (1977), pp. 762–772.
4. R. Cole, *Tight bounds on the complexity of the Boyer-Moore algorithm,* SIAM J. on Computing, 23 (1994), pp. 1075–1091.
5. L. Colussi, *Correctness and efficiency of pattern matching algorithms,* Information and Computation, 95 (1991), pp. 225–251.
6. R. Cole and R. Hariharan, *Tighter upper bounds on the comparison complexity of string matching,* in preparation. A preliminary version appeared in Proceedings of the 33rd Annual IEEE Symposium on the Foundations of Computer Science, (1992), pp. 600–609.
7. R. Cole, R. Hariharan, M. Paterson, U. Zwick, *Tighter lower bounds on the exact complexity of string matching,* SIAM J. on Computing, 24 (1995), pp. 30–45.
8. D. Dor and U. Zwick, *Selecting the median,* Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, (1995), pp. 28–37.
9. A. Ehrenfeucht and J. Mycielski, *Positional strategies for mean payoff games,* International Journal of Game Theory, 8 (1979), pp. 109–1139.
10. Z. Galil and R. Giancarlo, *On the Exact Complexity of String Matching: Lower Bounds,* SIAM J. on Computing, 6 (1991), pp. 1008–1020.
11. Z. Galil and R. Giancarlo, *On the Exact Complexity of String Matching: Upper Bounds,* SIAM J. on Computing, 3 (1992), pp. 407–437.
12. V. Gurvich, A. Karzanov, L. Khachiyan, *Cyclic games and an algorithm to find minimax cycle means in directed graphs,* USSR Computational Mathematics and Mathematical Physics, 28 (1988), pp. 85–91.
13. D.E. Knuth, J. Morris, V. Pratt, *Fast pattern matching in strings,* SIAM J. on Computing, 6 (1977), pp. 323–350.
14. A. Schönhage, M. Paterson, N. Pippenger, *Finding the median,* J. of Computer and System Sciences, 13 (1976), pp. 184–199.
15. U. Zwick and M. Paterson, *The complexity of mean payoff games,* Computing and Combinatorics (COCOON '95), LNCS 959 (Springer-Verlag 1995), pp. 1–10.