Searching for a Set of Correlated Patterns

Shmuel T. Klein, Riva Shalom and Yair Kaufman

Department of Computer Science

Bar Ilan University, Ramat-Gan 52900, Israel

Tel: (972–3) 531 8865 Fax: (972–3) 736 0498

tomi@cs.biu.ac.il shalom1@biu.013.net.il kaufmay@cs.biu.ac.il

Abstract: New algorithms for searching simultaneously for a set of patterns in a text are suggested, for the special case where these patterns are correlated and have a common substring. This is then extended to the case where it could be more profitable to look for more than a single overlap, and a problem related to the generalization of this idea is shown to be NP-complete. Experimental results suggest that for this particular application, the suggested algorithm yields significant improvements over previous methods.

Keywords: Pattern matching, multiple patterns, overlaps, NP-completeness.

1. Introduction

We concentrate in this paper on multiple pattern matching, in which a *set* of patterns $S = \{P_1, \ldots, P_k\}$, rather than a single one, is to be located in a given text T. This problem has been treated in several works, including Aho and Corasick [?], Commentz-Walter [?], Uratani and Takeda [?] and Crochemore et al. [?]. None of these algorithms assumes any relationships between the individual patterns. Nevertheless, there are many situations where the given strings are not necessarily independent.

Consider, for example, a large full text information retrieval system, to which queries consisting of terms to be located are submitted. If one wishes to retrieve information about `computers`, one might not want to restrict the query to this term alone, but include also grammatical variants and other related terms, such as `under-computerized`, `recomputation`, `precompute`, `computability`, etc. Using wild-cards, one could formulate this as `*comput*`, so that all the patterns to be searched for share some common substring.

In molecular Biology, multiple patterns sharing an overlap between them can be found in several situations. For example, in order to produce a certain protein, one needs to cut the gene coding it from the DNA and to transform it into bacteria that will manufacture that protein in a large amount. Cutting a certain gene from the DNA is done using restriction enzymes which cut the DNA sequence in certain nucleotide acid subsequences. As one does not know which sequences are situated upstream and downstream of the requested gene, one searches there for sequences of many restriction enzymes in order to detect the one that fits the needs. The sequences of the enzymes are often very similar and sometimes differ only in a single or a couple of nucleotide acids [?].

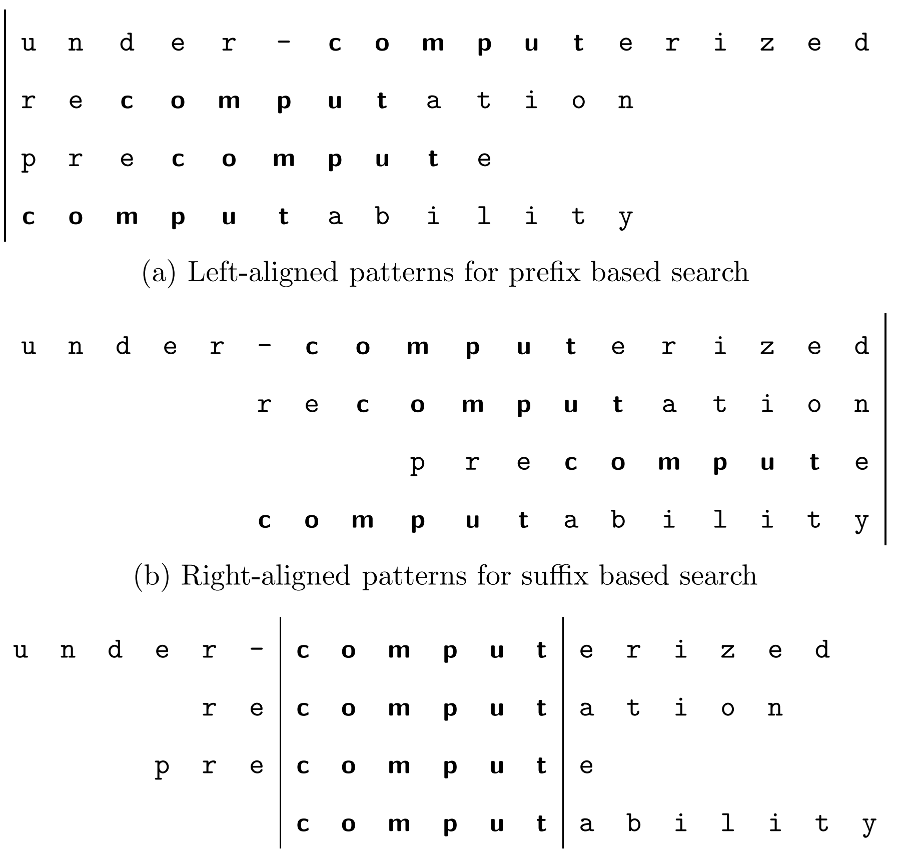
In the next section, we suggest new algorithms and relate them to previous work. Several alternatives are investigated, all based on finding one or more overlaps shared by all or a part of the patterns. Extending this idea further could possibly improve performance even more, but a related problem is shown to be NP-complete, suggesting that it might be hard to find the requested overlaps in the general case. Section 3 then brings some empirical experimental results.

2. Correlated Patterns

Navarro and Raffinot [?] classify the various algorithms for multiple pattern matching into three broad classes. In *prefix based* searches, the patterns are left aligned, as depicted in Figure 1a for the example strings mentioned above. This class includes the Aho Corasick (AC) algorithm [?], which is an extension of the Knuth, Morris and Pratt (KMP) algorithm [?] for a single pattern. In *suffix based* searches, the patterns are right aligned, as in Figure 1b. The Commentz-Walter (CW) [?] and the Wu Manber [?] algorithms belong to this class, both extending the basic Boyer-Moore (BM) method [?]. The third class is *factor based*, that is, treating arbitrary substrings of the patterns, and includes, e.g., DAWG matches [?]. We also consider general factors, but try to align the patterns so as to get the longest possible overlap, as can be seen in Figure 1c.

A known method for pattern matching is using a filter of the pattern that eliminates false occurrences in the text, leaving only potential occurrence locations that need to be verified. The algorithm of Amir, Benson and Farach [?], for example, uses a witness table and duels to rule out obvious mismatch locations, checking then the remaining locations using a so-called wave.

We suggest adapting the filtering concept to a set of correlated patterns, the basic idea being the following: if we can find a substantial overlap s, shared by all the patterns in the set, it is for s that we start searching in the text, using any single pattern matching algorithm, for example BM. If no occurrence of s is found, none of the patterns appears and we are done. If s does occur $t > 0$ times, it is only at its t locations that we have to check for the appearance of the set of prefixes of s in the set of patterns and of the corresponding set of suffixes. This can be done locally at the t positions where s has been found, e.g., with the AC algorithm, but with no



(a) Left-aligned patterns for prefix based search

(b) Right-aligned patterns for suffix based search

(c) Overlap alignment

FIGURE 1: *Possible alignments of a set of patterns*

need to use its *fail* function.

More formally, let the set S consist of patterns P_i, where $P_i = l_i\, s\, r_i$, and l_i and r_i are the (possibly empty) prefixes and suffixes of P_i which are left after removing the substring s. For our example, $s =$`comput`,$\{l_i\} = \{$`under-`, `re`, `pre`, $\Lambda\}$, where Λ denotes the empty string, and $\{r_i\} = \{$`erized`, `ation`, `e`, `ability`$\}$. Denote also the length of P_i by m_i and the total length $\sum_{i=1}^{k} m_i$ by M. The algorithm starts by identifying s, the longest common substring shared by $P_1, \ldots, P_k$. The search algorithm is then given by:

```
Overlap_Matching($s, S$)

search for $s$ in text $T$ using KMP or BM
for each $i$ such that $s$ is found starting at position $i$
        check at position $i + |s| - 1$ for an occurrence of an element of $\{r_j\}$
                using an AC automaton
        for each matching $r_j$ found
                check if $l_j$ matches $T$ at position $i - |l_j|$
                if yes, declare match at $i - |l_j|$
```

The dominant part of the time complexity will generally be for the search of s, which can be done in time $O(|T|)$. Applying the AC automaton is not really a search, since it is done at well-known positions. Its time complexity is therefore bounded $O(t \cdot \max\{m_i\})$, where t is the number of occurrences of the overlap s. Only in case the occurrences of s are so frequent that the potential positions of the patterns cover a large part of the string T may $O(t \cdot \max\{m_i\})$ be larger than $O(|T|)$, but this will rarely occur for a natural language input string.

There are also other cases for which our approach is not necessarily superior to previously suggested ones. Generally it will be true that the longer the overlap, the smaller the probability of its occurrence, and thus the faster will it be to search for the entire set, but there can obviously be exceptions to this rule. Moreover, the main parameter influencing the processing time of our algorithm is the length of the overlap, whereas the lengths of the strings themselves play only a secondary role. On the other hand, algorithms based on backward searches, like CW, generally increase their efficiency with the lengths of the patterns. In the special case of a set of very long strings having only a relatively small common overlap, CW could thus be preferable.

One should thus add some rule of thumb before applying the algorithm we suggest. For example, consider the ratio between the length of the overlap s and the average length of the patterns in the given set S: only if this ratio is larger than some predetermined threshold should our algorithm be applied, otherwise prefer CW or one of the other alternatives.

2.1 Single overlap giving full coverage

Our main concern is with the matching algorithm for multiple patterns. The longest overlap is used as a filter of the dictionary, thus it must be found without increasing the overall time complexity of the search.

An efficient way to achieve this is to use an elegant linear time algorithm due to Hui [?]. It suggests building a generalized suffix tree [?] for the patterns of the dictionary (in such a compacted tree, every internal node actually stands for an overlap of two or more patterns). In the next steps, the tree is traversed in postorder and some constant time computations are performed for every internal node, including applying a *Lowest Common Ancestor* [?] procedure. At the end of the process, each internal node v includes information about its frequency $f(v)$ in the dictionary, i.e., the number of distinct patterns of which the string represented by the node is a subpattern. In order to find the longest overlap giving full coverage, one simply traverses the tree once again and selects a string of maximal length among those with frequency $f(v) = k$. The total time for finding the required overlap is thus $O(M)$.

Consider now as example the dictionary $S = \{P_1 = \texttt{dxyz}$, $P_2 = \texttt{wdxyza}$, $P_3 = \texttt{bcdxyzw}$, $P_4 = \texttt{bcdzw}\}$, showing a major deficiency of the suggested algorithm.

The corresponding list of overlapping substrings is given in Table 1. Note that the longest substring shared by all the patterns in this set is just a single character. One can of course circumvent the case, which is even worse from our point of view, when the longest common substring is empty, by invoking then the standard AC routine. However, if there is a non-empty string s, but it is too short as in this example, it might occur so often that the benefit of our procedure could be lost.

overlap	# patterns	length
`d`	4	1
`z`	4	1
`dxyz`	3	4
`xyz`	3	3
`yz`	3	2
`w`	3	1
`bcd`	2	3
`cd`	2	2
`zw`	2	2

TABLE 1: *List of overlaps*

In the above example, the string `dxyz` is shared by only three of the four elements of the dictionary S, but its length is much longer than the string shared by all the elements. This suggests that it might be worthwhile not to insist on having the overlap cover the entire dictionary, but maybe to settle for one shared not by all, but at least by a high percentage of the patterns, which may allow us to choose a longer overlap. An alternative approach could be to look for two or more substrings $s_1, s_2, \ldots$, each longer than s and each being shared by the patterns of some proper subset of S, but which together cover the entire dictionary. For our example we could, e.g., use the pair of substrings $\{\mathtt{dxyz}, \mathtt{bcd}\}$. In Information Retrieval applications, such an approach could be profitable in case the query consists of the grammatical variants of two or more terms, or in case of irregularities, as in the set {`go`, `goes`, `undergoing`, `went`}. In the next subsection we explore the details of the first alternative. The case of choosing an overlap pair is considered in Section 2.3 and partition into more than two subsets is discussed in Section 2.4.

2.2 Single overlap giving partial coverage

We have seen that a single overlap is not always efficient. There is a tradeoff, for each potential overlap s, between its length $|s|$ and the number of patterns it covers $f(s)$. We shall choose an overlap that maximizes the product of these two factors. The products can be easily evaluated by adapting Hui's algorithm, without changing its complexity, to store $|s| \cdot f(s)$ at the node corresponding to string s. The overlap s_0 maximizing the product will be used as the chosen overlap in the Overlap_Matching algorithm, but the induced dictionary includes only the patterns covered by s_0. For the rest, the algorithm can be applied recursively. This yields the Greedy_Overlap_Matching defined by:

```
Greedy_Overlap_Matching($S$)

build a generalized suffix tree for $S$
for every potential overlap $s$ compute $|s| \cdot f(s)$
select $s_0$ maximizing the above product
define $S'$ to be the subset of $S$ covered by $s_0$
apply Overlap_Matching($s_0, S'$)
recursively call Greedy_Overlap_Matching($S - S'$)
```

2.3 Overlap pair giving full coverage

The greedy algorithm above tried to increase the lengths of the overlaps that will be searched for, reducing thereby the expected overall search time. While improving over the strict approach seeking for an overlap covering the entire dictionary, its recursive nature may turn out to be wasteful: it may find a pattern covering a good part of the dictionary, but splitting the remaining patterns into many small subsets, so that the number of recursive calls might be $\Omega(k)$. Since for each call the text is scanned, this may yield an $\Omega(k|T|)$ algorithm, which is prohibitive.

This leads to the idea of changing again the target: instead of looking for a single overlap covering as much of the dictionary as possible, seek a *pair* of overlaps s_1, s_2 such that each pattern in S has at least one of the two s_i as substring; among all such pairs, choose the one with maximal $|s_1| + |s_2|$. We further demand that each of these two be longer than a single overlap shared by all patterns, if it exists. If such a pair can be found, let S_1 and S_2 be the subsets of S covered by s_1 and s_2 respectively, and apply Overlap_Matching(s_1, S_1) and Overlap_Matching$(s_2, S_2 - S_1)$ (the $S_2 - S_1$ comes to avoid unnecessary comparisons in case $S_1 \cap S_2$ is not empty).

To find the pair of overlaps, a list of all the overlaps in the dictionary is created, using as before the suffix tree built by [?]. The overlaps are then sorted by decreasing length. If there exists an overlap of length ℓ shared by all patterns of the dictionary, we do not consider overlaps shorter than ℓ in our search for a covering pair, so overlaps shorter than ℓ can be deleted from the list. A special case to be dealt with is when there is an overlap s_1 with frequency $k - 1$ in the list, i.e., a substring shared by all but one of the patterns. In that case the only remaining pattern can be considered itself as the complementing overlap s_2. For the other cases we proceed as follows.

Define a binary matrix $\mathcal{C}$ whose rows $\mathcal{C}[x,*]$ represent the overlaps of frequency $k-2$ and lower, sorted by decreasing length, and whose k columns stand for the patterns of the dictionary. $\mathcal{C}[i,j]$ will be set to 1 if and only if the ith overlap is a substring of P_j. The matrix can be used to check whether the two overlaps corresponding to rows x and y cover the entire dictionary: this will be the case if and only if $\mathcal{C}[x,*] \ \vee \ \mathcal{C}[y,*] = 1^k$, where 1^k stands for the bit-string consisting of k 1s, and $\vee$ is the bitwise OR operation.

Let R be the number of rows in $\mathcal{C}$. The algorithm processes then the rows of the matrix in a double loop, ORing pairs of rows. Among the pairs giving 1^k, the pair with maximum combined length of the corresponding overlaps is chosen. Formally, denote by $s(i)$ the overlap of row i, $1 \le i \le R$, and use an array of flags $flag[i]$ to indicate whether $s(i)$ has already been part of a pair yielding 1^k; if so, row i can be skipped, because the rows are ordered by non-increasing lengths of the overlaps. The formal algorithm, returning the pair (m_1, m_2) of the indices of the requested overlaps, is given below.

The number R of possible overlaps is bounded by the number of possible different substrings, which is the number of internal nodes in the generalized suffix tree, so $R = O(M)$ in the worst case, but will often be much less. Finding all the overlaps can be done in $O(M)$ time, and their sorting in $O(M \log M)$.

The size of the matrix is $O(Mk)$, so this bounds its construction time, but the nested loops may take $\theta(kM^2)$ in the worst case. This might possibly be improved using bit-parallelism for small k: on 8-byte machines, 64 bitwise OR operations can be done in a single machine instruction, but we are still left with at least $\theta(M^2)$. This might seem expensive if M is large, but recall that M will generally be small relative to T, so $\theta(M^2)$ in the pre-processing stage might be tolerated.

If bit-parallelism can be used and k is small, an alternative way for finding the

```
FIND_OVERLAP_PAIR

$m_1 \longleftarrow m_2 \longleftarrow s(0) \longleftarrow 0$
for $x \longleftarrow 1$ to $R$   $flag[x] \longleftarrow False$
for $x \longleftarrow 1$ to $R$
    if not $flag[x]$ then
        for $y \longleftarrow x+1$ to $R$
            if not $flag[y]$ AND $\mathcal{C}[x] \vee \mathcal{C}[y] = 1^k$ AND $|s(x)| + |s(y)| > |s(m_1)| + |s(m_2)|$ then
                $(m_1, m_2) \longleftarrow (x, y)$
                $flag[x] \longleftarrow flag[y] \longleftarrow True$
                exit inner loop
        end for $y$
end for $x$
```

matching pairs is as follows: consider the k columns $\mathcal{C}[*, j]$ of the matrix $\mathcal{C}$, each corresponding to one of the patterns P_j for $j = 1, \ldots, k$. To find the index y of the overlap forming the requested matching pair with the overlap indexed x, consider the binary complement of the x-th row of the matrix, $\overline{\mathcal{C}[x, *]}$, and denote the set of the indices of its 1-bits by $\mathcal{I}$. Now perform

$$\mathcal{V} = \bigwedge_{j \in \mathcal{I}} \mathcal{C}[*, j],$$

that is, all the columns of the matrix $\mathcal{C}$ corresponding to indices in $\mathcal{I}$ are ANDed together, and the result is the column vector $\mathcal{V}$; the indices of the 1-bits of $\mathcal{V}$ are those forming with x a covering pair, so we get that y is the smallest such index of a bit position in $\mathcal{V}$ containing a 1-bit, since the overlaps are ordered by non-increasing lengths. The length of the columns is $\theta(M)$ so the ANDing to get $\mathcal{V}$ takes up to $\theta(kM)$. Since this is done for each row x, we again get $\theta(kM^2)$. As the columns might be much longer than the rows, bit-parallelism can yield more savings with this approach.

2.4 The Overlap Partition problem

A natural extension of the above ideas, in case there is no single or pair of overlaps, would be to look for a set of strings $s_1, s_2, \dots s_m$ and a corresponding partition of the set of patterns S into disjoint subsets $S_1, \dots, S_m$, such that the patterns in S_i have s_i as longest common substring. This is reminiscent of the Partition problem, which is NP-complete. We show below that a problem similar, though for technical reasons not exactly identical, to ours is also NP-complete, which suggests that it is probably very hard to find an optimal solution and thereby justifies heuristics as those suggested above. Consider the following decision problem.

The Overlap Partition Problem (OPP): Let S be a set of strings over an unbounded alphabet Σ, and let m, w_1, w_2, w_3 be some integer constants, with $m \geq 2$. Find a partition of S into at least m subsets $\{S_1, S_2, ..., S_m, ...\}$, such that $|S_i| \geq 2$ and the strings in S_i share some overlap v_i of length at least w_1 for S_1, at least w_2 for S_2, and at least w_3 for the other S_j, $j > 2$. Moreover, the overlap strings v_i should themselves be non-overlapping.

The rationale behind this non-overlapping condition is that if v_i and v_j overlap, we could have merged the sets S_i and S_j and still get an non-empty overlap, though a shorter one. The non-overlapping requirement of the v_i should be understood in the following sense. The selection of an overlap v_i in fact induces a certain alignment of the elements of S_i, which then permits a sequential numbering of the character positions in the set. The leftmost position of the set will be defined as corresponding to the longest prefix of v_i in the strings belonging to S_i. For example, if $S_i = \{\texttt{abc}\alpha\beta\gamma\texttt{de}, \texttt{fg}\alpha\beta\gamma\texttt{hijk}\}$, the overlap is $v_i = \alpha\beta\gamma$, the leftmost position corresponds to a, there are 10 positions in the set, and the overlap v_i starts at position 4. When comparing v_i with v_j, we first align the two sets S_i and S_j according to their leftmost positions, and then check whether the corresponding positions of the overlaps v_i and v_j have a non-empty intersection. If so, we require that v_i and v_j should differ on their overlapping part. To continue the above example, suppose $S_j = \{\texttt{lmnop}\delta\epsilon\texttt{qr}, \texttt{st}\delta\epsilon\texttt{uvwx}\}$, the alignments are then depicted in Figure 2.

position	1	2	3	4	5	6	7	8	9	10	11
S_i	a	b	c	α	β	γ	d	e			
		f	g	α	β	γ	h	i	j	k	
S_j	l	m	n	o	p	δ	ϵ	q	r		
				s	t	δ	ϵ	u	v	w	x

FIGURE 2: *Alignment of overlaps*

In this case, v_i and v_j overlap in position 6, but since $\gamma \neq \delta$, such an overlap is permitted. If v_j were $\gamma\epsilon$ instead of $\delta\epsilon$, v_i and v_j would violate the non-overlap requirement. Note also that the definition of OPP does not require the overlaps to be consecutive substrings.

In Computational Biology, multiple alignment of strings is often considered with a score called Sum of Pairs [**?**, **?**] and related problems are shown to be NP-complete, however, due to the different definition of the scoring, our problem is quite different and its complexity does not follow from that of the Sum of Pairs based problems.

Theorem. *The Overlap Partition Problem is NP-Complete.*

Proof: After guessing the partition $S_1, S_2, \ldots$, one can check in polynomial time that all elements of S are members of some subset S_i, that each S_i contains an overlap of the proper length and that these overlaps are themselves non-overlapping, thus OPP $\in$ NP. For the reduction, we show that CLIQUE $\propto$ OPP, where we use the standard notation $X \propto Y$ to denote that X is reducible to Y.

Construction: Consider a general instance of the Clique problem: a graph $G = (V, E)$, with $V = \{1, 2, \ldots, n\}$, and a constant K, $K > 2$. The alphabet Σ over which the strings in S will be defined consists of the characters a, b, 1 and $2n$ different symbols 0_i. The set of strings S will consist of $2n$ strings X_i and Y_i, $1 \leq i \leq n$, and each of the different zero-symbols will only appear in a unique string X_i or Y_i. We shall therefore, for the ease of description, drop the subscripts of the zeros, refer only to a quaternary alphabet $\Sigma = \{a, b, 1, 0\}$, but remember that 0s belonging to different strings never match.

Each pair of strings X_i and Y_i corresponds to one of the vertices of V and is divided into four fields, A, B, C and D, defined as follows.

Field A: This field will serve to enforce alignment between the elements of S_1. Its length is $2n^2$ and it is defined as

$$A = \begin{cases} (a^n 0^n)^n & \text{for } X_i \\ 0^{2n^2} & \text{for } Y_i \end{cases}$$

Field B: This is the main field relating to the graph and it is of length n characters, B= $b_{i,1} \cdots b_{i,n}$, defined by:

for X_i:

$$b_{i,j} = \begin{cases} 1 & \text{if } j = i \vee (i,j) \in E \\ 0 & \text{otherwise,} \end{cases}$$

for Y_i:

$$b_{i,j} = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise.} \end{cases}$$

Field C: This field contains n subfields $C_{i1}, C_{i2}, ..., C_{in}$. Subfield C_{ij} is of length $4n^2 + n(j+1) + 1$ defined as:

for X_i:

$$C_{ij} = \begin{cases} 0^{4n^2+n(j+1)+1} & \text{if } j \neq i \\ 1(1^{n+1}0^{n+i})^n 0^{2n^2} & \text{if } j = i \end{cases}$$

for Y_i the definition looks identical, but recall that the 0 symbol in Y_i is different from (does not match) that of X_i.

Field D: This field will serve to enforce alignment between the elements of S_2. Its length is $2n^2 + n + 1$ and it is defined as

$$D = \begin{cases} 0^{(2n+1)n+1} & \text{for } X_i \\ b(b^{n+1}0^n)^n & \text{for } Y_i. \end{cases}$$

To complete the construction, the constants are set as: $m = n - K + 2$, requiring a partition into at least two subsets, and $m = 2$ only if the size of the clique is $K = n$, i.e., a full graph is required; $w_1 = n^2 + K$, $w_2 = n^2 + n + 1$ and $w_3 = n^2 + n + 2$.

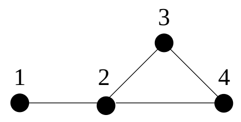


Figure 3: *Example graph G*

The construction is polynomial in the size of the graph, as for vertex i we build two strings of length $O(n^3)$, so altogether, time and space are bounded by $O(n^4)$.

The following example should clarify these definitions: for an instance of Clique with $K = 3$ and the graph G in Figure 3, the construction is given in Figure 4.

	A	B	C	D
$X_1 =$	$(a^4 0^4)^4$	1100	$1(1^5 0^5)^4 0^{32}, 0^{77}, 0^{81}, 0^{85}$	0^{37}
$Y_1 =$	0^{32}	1000	$1(1^5 0^5)^4 0^{32}, 0^{77}, 0^{81}, 0^{85}$	$b(b^5 0^4)^4$
$X_2 =$	$(a^4 0^4)^4$	0111	$0^{73}, 1(1^5 0^6)^4 0^{32}, 0^{81}, 0^{85}$	0^{37}
$Y_2 =$	0^{32}	0100	$0^{73}, 1(1^5 0^6)^4 0^{32}, 0^{81}, 0^{85}$	$b(b^5 0^4)^4$
$X_3 =$	$(a^4 0^4)^4$	0111	$0^{73}, 0^{77}, 1(1^5 0^7)^4 0^{32}, 0^{85}$	0^{37}
$Y_3 =$	0^{32}	0010	$0^{73}, 0^{77}, 1(1^5 0^7)^4 0^{32}, 0^{85}$	$b(b^5 0^4)^4$
$X_4 =$	$(a^4 0^4)^4$	0111	$0^{73}, 0^{77}, 0^{81}, 1(1^5 0^8)^4 0^{32}$	0^{37}
$Y_4 =$	0^{32}	0001	$0^{73}, 0^{77}, 0^{81}, 1(1^5 0^8)^4 0^{32}$	$b(b^5 0^4)^4$

$$m = 3 \qquad w_1 = 19 \qquad w_2 = 21 \qquad w_3 = 22$$

FIGURE 4: *Construction corresponding to graph G and $K = 3$*

We now turn to the proofs showing that we have a valid reduction.

Claim ($\Rightarrow$): *If there is a clique of size K in G, then there is a valid partition.*

Proof: Suppose we have a clique of size K in G, let $I = \{i_1, \ldots, i_K\}$ denote the set of its vertices. Let $J = V \backslash I = \{j_1, \ldots, j_{n-K}\}$ denote the complementary set. The partition will be $S_1 = \{X_i \,|\, i \in I\}$, $S_2 = \{Y_i \,|\, i \in I\}$ and $S_t = \{X_{j_{t+2}}, Y_{j_{t+2}}\}$, for $2 < t \leq m$, that is, S_1 contains K X_i's and S_2 contains K Y_i's and their indices correspond to the vertices of the clique; each of the other S_t, if there are any, contains exactly two elements, one X_j and one Y_j, both with the same index, and these indices correspond to vertices not in the clique.

To check the validity of this partition, note that S_1 and S_2 have at least two elements as we require $K > 2$. All strings in S_1, when left aligned, share an overlap in field A of length n^2 (all the a characters). They also have K 1's in field B that overlap too, yielding a total overlap of size $n^2 + K = w_1$, as required. Fields C and D do not match for these strings.

All strings of S_2 cannot overlap in field A due to the definition of the different 0s. Moreover, they cannot have anything in common in field B (or C), as there are 1s only in the position (or subfields) corresponding to the index of the string. But all elements of S_2 have the same field D, overlapping in their b's. Therefore, the overlap of S_2 is of size $n^2 + n + 1 = w_2$.

In each S_t, for $t > 2$, if it exists at all (in the special case $K = n$, S_1 and S_2 are the only sets in the partition), we have X_i and its corresponding Y_i so there is a common 1 at the ith position of field B. In addition, field C as a whole is identical for (the non-zero positions of) both of them, so they share an $n^2 + n + 2$ overlap, which is equal to w_3.

There is no overlap between the overlaps: S_1 and S_t both have field B as part of their overlap, but not at the same positions due to the fact that the strings in S_t correspond to vertices that are not in the clique and the strings in S_1 correspond to vertices that are. The overlap in S_2 is exclusively from field D thereby avoiding any overlap with S_1 and S_t, $t > 2$. There can be no overlap between the overlaps v_t of different S_t, for $t > 2$, since v_t is based on field C, and these fields have nothing in common for different t. ∎

For the above example, the partition we get is:

$S_1 = \{X_2, X_3, X_4\} \rightarrow$ the overlap is $(a^4)^4 111$ with length $19 = w_1$.

$S_2 = \{Y_2, Y_3, Y_4\} \rightarrow$ the overlap is $b(b^5)^4$ with length $21 = w_2$.

$S_3 = \{X_1, Y_1\} \quad \rightarrow$ the overlap is $11(1^5)^4$ with length $22 = w_3$.

Claim ($\Leftarrow$): *If there is a valid partition, then there is a clique of size K in G.*

Observation 1 *All X_i and Y_i are aligned.*

Proof: Our definition of an overlap between two strings allowed the strings to be shifted to maximize the overlap length as in the examples in Figure 2. For the current construction, however, we show that all the strings ought to be aligned. Consider all possible subsets in the partition, subsets consisting only of X_i strings (type 1), similar subsets of Y_i strings (type 2), and mixed subsets, including both X and Y strings (type 3). If we shift some strings in a subset of X_i strings, their C field overlap can consist of at most $1 + \sum_{r=2}^{n+1} r = \frac{1}{2}(n^2 + 3n + 2)$ characters, and even that only for strings X_i and X_{i+1}; the A field overlap gets smaller by, at least, n characters for any shift, so the length of the overlap cannot be more than $\frac{1}{2}(n^2 + n + 2)$. But $m \geq 2$ and $|S_i| \geq 2$ so that we consider only instances of Clique for which $n \geq 4$; for such n, the bound on the length of the overlap $\frac{1}{2}(n^2 + n + 2)$ is always smaller than n^2, which is smaller than each of w_1, w_2 and w_3.

Similarly, in a set of Y_i strings, any shift will reduce the overlap by at least n characters in field D, the C field, again, can contribute at most $\frac{1}{2}(n^2 + 3n + 2)$ overlapping characters, so again, no shift can increase the overlap length, on the contrary it will decrease it.

For the mixed sets of type 3, if a set includes X_i and Y_j for $i \neq j$, there is at most an overlap of 1 character in the B field of X_i and Y_j, and if we try to maximize the overlap by shifting to align the 1s is the C field, the maximal match is again $\frac{1}{2}(n^2 + 3n + 2)$, which is less than any w_j. If the set includes X_i and Y_i, any shift will decrease the match in the C field. ∎

Observation 2 *No subset includes both X_i and Y_j, for $i \neq j$.*

Proof: If such a pair is included in one of the subsets, their A, B and D fields are totaly distinct and the C field can contribute at most $\frac{1}{2}(n^2+3n+2)$ overlapping characters, which is smaller than any w_i, thus such a subset could be not a valid one in our partition. ∎

As a consequence, the only possible subsets of the partition including both X_is and Y_js are of size two and consist of X_i and its corresponding Y_i.

Observation 3 *A set of the partition including two or more X_i strings, has an overlap of at most n^2+n characters (in fields A and B).*

Proof: The possible overlap between different X_i strings is due to fields A and B, since field C is unique for each pair X_i, Y_i and the D field consists only of 0s for X strings. Field A yields an overlap $(a^n)^n$ of length n^2 between any two X_i strings, and field B is of length n. ∎

Observation 4 *A set of the partition including two or more Y_i strings, has an overlap of size at most n^2+n+1 (in field D).*

Proof: Different Y_i strings cannot overlap in field A. They cannot match in field B since it has 1's only in the position of the index i, and field C is unique for each pair of strings X_i, Y_i. Therefore the only overlap possible is in field D which is identical for all Y_i, yielding an overlap of length n^2+n+1. ∎

As a consequence of Observations 3 and 4, every set S_t, for $t > 2$, consists of exactly one X_i and one Y_j. From Observation 2 follows that $i = j$, i.e, they are strings with the same index.

S_2 cannot include only X strings due to Observation 3 and the request of overlap size at least n^2+n+1. Note that $2(n-K)$ strings are already included in S_t for

$t > 2$, leaving $2K \geq 6$ strings to be partitioned into S_1 and S_2. If S_2 includes only one X_i and its corresponding Y_i, then there are at least two Xs and two Ys in S_1, but any set including at least two Xs and a Y can have an overlap of at most n^2+1 (field A and one character in B), which is less than w_1. If S_2 includes a single X and two or more Ys, the overlap can be of length at most n^2 (field A only), less than w_2. It follows that S_2 cannot include Xs at all, and is thus of the form $S_2 = \{Y_j\}_{j \in I}$ for some subset $I \subseteq \{1, \ldots, n\}$ of indices.

It follows that the remaining strings, which are $\{X_j\}_{j \in I}$, form the set S_1. By assumption, the overlap v_1 in this set is of size at least $n^2 + K$. Since different Xs have no overlap in fields C and D, and exactly an n^2 overlap in field A, there must be an overlap of at least K in the B fields.

Observation 5 *Let $b_1 \cdots b_n$ be the B field of the overlap v_1. If $b_i = 1$, then $X_i \in S_1$.*

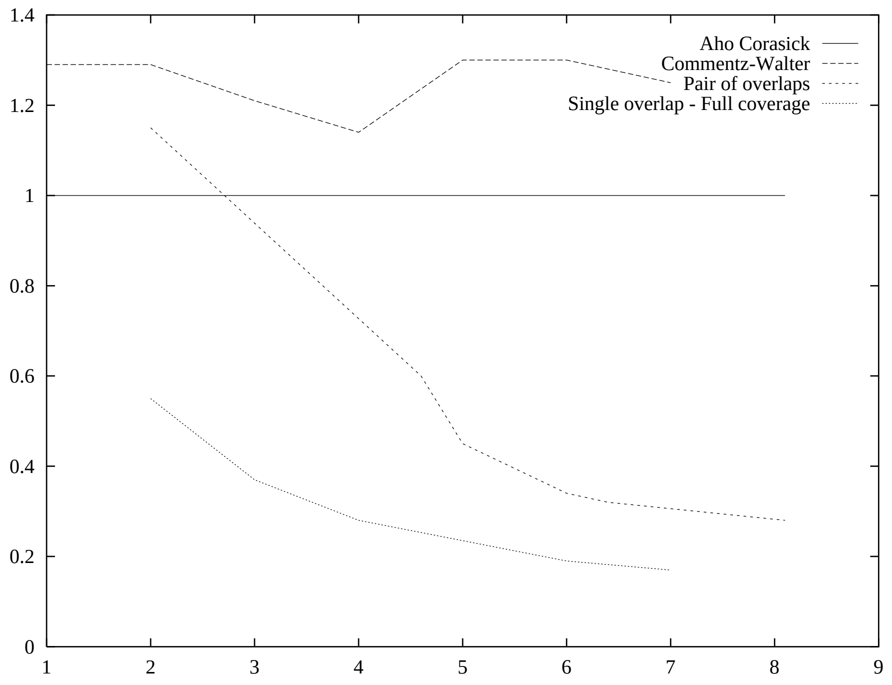
Proof: Assume $X_i \notin S_1$, then $X_i \in S_t$, for some $t > 2$, and position i of field B is part of that overlap of S_t. However, this fact implies that position i of field B in S_t is identical to that of S_1, which contradicts the non-overlap requirement of the overlaps of different subsets. ∎

Consider now any two indices $i, j \in I$. From Observation 5 we know that both X_i and X_j are in S_1. Since positions i and j of field B of the overlap contain 1s, this is true in particular also for string X_i, which means, by the construction of field B, that the edge $(i, j) \in E$. Since this is true for any pair of indices in I, the vertices corresponding to these indices form a clique of size $\geq K$. ∎

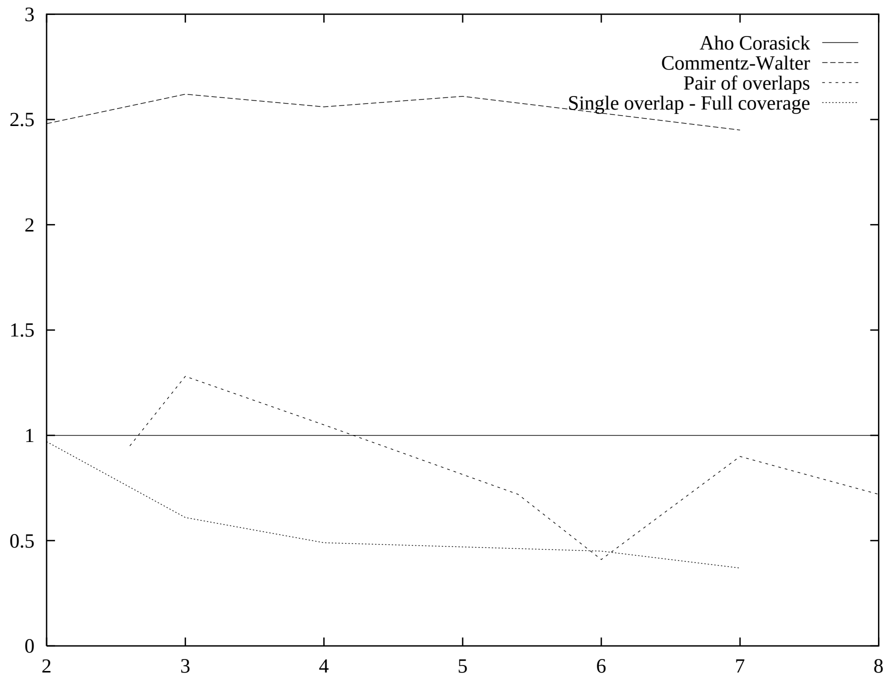
3. Experimental Results

The complexity of the suggested search algorithm is first $O(|T|)$ for finding the occurrences of the overlap(s), and then $O(tM)$ to check for possible occurrences of the prefixes and suffixes next to the t positions at which the overlap has been found. There is thus no improvement of order of magnitude relative to all the algorithms that do not assume any correlation within the set of patterns. But in the AC algorithm, for instance, every character of the text is inspected, whereas when searching for a common overlap, the search can be performed sub-linearly, for example using BM. The remaining $O(tM)$ will often be much smaller than $O(|T|)$, so that the overall performance is improved. To get a general feeling of how the algorithms behave in some real-life applications, we ran a set of tests on several text and DNA files. The tests are not necessarily representative, as the number of occurrences of the chosen overlap in a given application may be much lower, improving the overall search time.

The chosen texts were several plays by Shakespeare, as well as three DNA strings (human hemoglobin, Caenorhabditis elegans and human chromosome CDKN1C) to check the influence of the size of the underlying alphabet. A random choice of the patterns to be searched for does not yield interesting results, since the common overlap of randomly chosen terms will often be empty or very short. We thus proceeded as follows: a random term (the *seed*) of length ℓ was chosen from each text. This terms was then extended by prefixes and suffixes of varying lengths to form a dictionary S. The tests were repeated with strings of seed length $\ell = 1, \ldots, 7$, yielding different dictionaries, each containing between 8 and 10 keywords for the text files. For the DNA files, 2 dictionaries of 10 keywords for each seed length $\ell = 2, \ldots, 7$ were built.

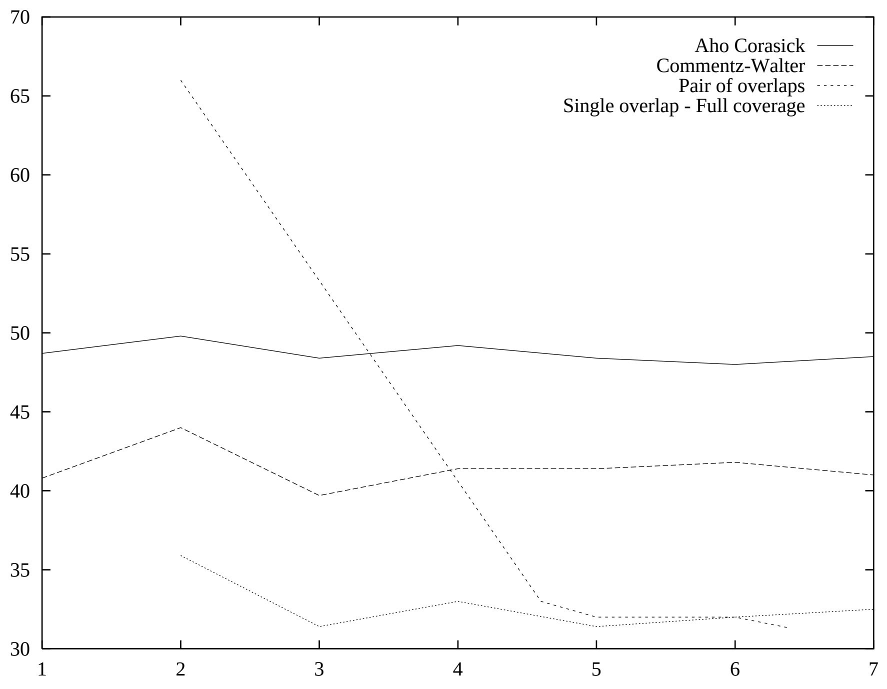


(a) Text files

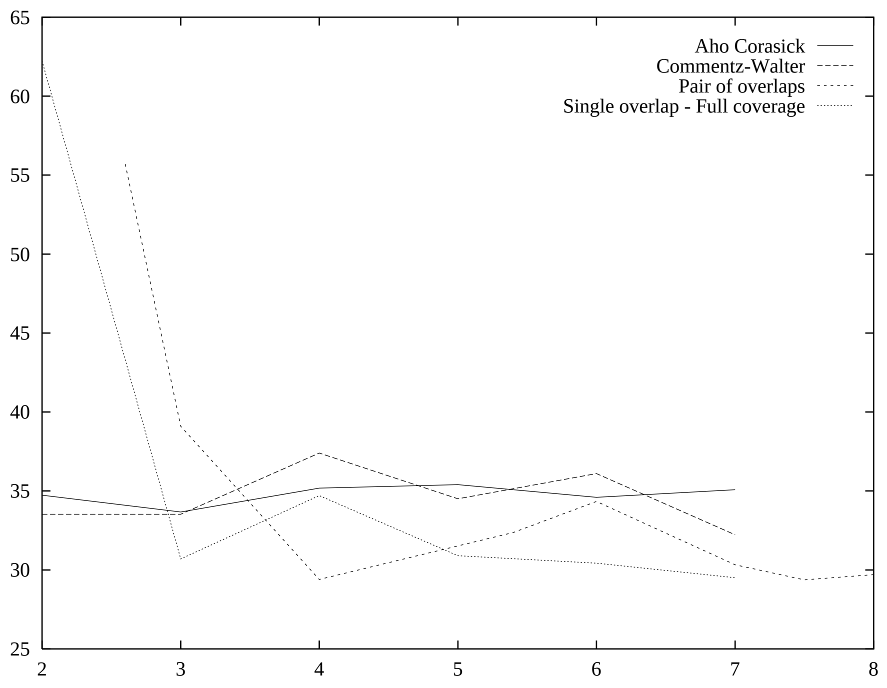


(b) DNA strings

FIGURE 5: *Comparative performance of matching algorithms — comparisons*



(a) Text files



(b) DNA strings

FIGURE 6: *Comparative performance of matching algorithms — time (ms)*

As a measure for the efficiency, we defined a *rate* as the number of symbol comparisons divided by the length of the text. The Aho Corasick algorithm served as benchmark, yielding always a rate of 1. The searches were performed with the Commentz-Walter algorithm, the OVERLAP_MATCHING, denoted SO (Single Overlap) in the sequel, and with the algorithm using a Pair of Overlaps (PO), and the averaged values are displayed in Figure 5. The graphs give the rate, for each of the algorithms, as a function of the overlap size (for PO, the weighted lengths of the overlaps was taken as basis). In addition, empirical timing results were recorded, and these are displayed in Figure 6. Results for the GREEDY_OVERLAP_MATCHING are not displayed, as they were mostly inferior to the others, especially for the real timing results.

The rate of AC is always one as the algorithm actually goes over the text exactly once. The rate of CW is also quite close to a constant, for real words between 1.14 and 1.3, and for DNA files between 2.45 to 2.62. The difference between the file types is due to the fact that CW is based on the Boyer-Moore algorithm, which is faster for larger alphabets. Interestingly, we got faster execution of CW than AC, while the number of comparisons was lower for AC than for CW.

As can be seen, for both text and DNA files, the algorithms we proposed have a better rate than CW, and outperform also AC for large enough overlaps. Moreover, the rate seems to be decreasing with the overlap size, and rates as low as 0.5 on the average can be reached already for relatively short overlaps of size 2–4.

As to execution time, all the algorithms require about the same time for DNA files, with the new overlap algorithms consuming slightly less time than the AC and CW algorithms. The differences are more evident for regular text files. The performance of AC is the worst, whereas SO and PO improve even over the CW algorithm (PO only for overlaps of size at least 4).

The SO algorithm seems mostly superior to the others, both according to the number of character comparisons and for real timing measurements, but in fact such a comparison is meaningless because the algorithms will be applied in different situations: if there is a single overlap of length ℓ, it makes obviously no sense to use a pair or more overlaps with the same weighted length.

4. Conclusion

A new algorithm for the efficient search of a set of patterns has been presented, for the special case in which these patterns are correlated in the sense of sharing some common substring. This may be useful for certain applications in Information Retrieval and Molecular Biology. We have dealt with finding the overlaps and how to use them for improved search. The analysis and experimental results suggest that for the special case under consideration, the new algorithms may significantly improve performance over previous methods.

For future work, we consider transferring the ideas into the compressed domain, searching for a set of compressed patterns in a compressed text. The underlying alphabet is then binary, so that longer common substrings might be expected, which increases the attractivity of the suggested method.

Acknowledgement: The authors wish to thank three anonymous referees for their helpful comments.

References

[1] A.V. Aho and M.J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Comm. of the ACM* 18 333–340, 1975.

[2] A. Amir, G. Benson and M. Farach. Alphabet-Independent Two Dimensional Matching. *Proceedings, 24th Annual ACM Symposium on the Theory of Computation (STOC)* pp. 59-68, 1992.

[3] A. Amir, M. Farach, Z. Galil, R. Giancarlo, K. Park, Dynamic Dictionary Matching.*Journal of Computer and System Sciences*, 49 208–222, 1994.

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

[5] B. Commentz-Walter. A string matching algorithm fast on the average. *Proc. 6th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Sci. Springer, Berlin,* 118–132, 1979.

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

[7] M. Crochemore, T. Lecroq. Pattern Matching and Text Compression Algorithms, *The Computer Science and Engineering Handbook*, A.B. Tucker Jr., ed., CRC Press, Boca Raton, 1997, Chapter 8, 162–202.

[8] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, W. Rytter. Fast practical multi-pattern matching. *Information Processing Letters* 71 107–113, 1999.

[9] I. Elias. Setting the Intractability of Multiple Alignment, *Proc. of the 14th Ann. Int. Symp. on Algorithms and Computation (ISAAC'03), LNCS* 2906, 352–363, 2003.

[10] R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. *In Proc. 1st South American Workshop on String Processing (WSP 1993)*, pages 57-76, September 1993.

[11] D.E. Knuth, J.H. Morris and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comp.* 6 323 -350, 1977.

[12] L. C. K. Hui. Color Set Size Problem with Applications to String Matching. *Combinatorial Pattern Matching, LNCS* 644 : 230-243, 1992.

[13] W. Just. Computational Complexity of Multiple Sequence Alignment with SP-Score. *J. of Computational Biology* 8 615–623, 2001.

[14] U. Manber, G. Myers. Suffix arrays: a new method for on-line string searches, *Proc. ACM-SIAM SODA* 319–327, 1990.

[15] H. M. Martinez. An efficient method for finding repeats in molecular sequences, *Nucleic Acids Research* 11(13) 4629-4634, 1983.

[16] C. K. Mathews, K. E. van Holde and K. G. Ahern. *Biochemistry.* Addison Wesley Longman Print, 2000.

[17] G. Navarro and M. Raffinot. *Flexible Pattern Matching in String - Practical on-line search algorithms for text and biological sequences*. Cambridge University Press, 2002.

[18] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal of Computing*, 17:1253-1262, 1988.

[19] N. Uratani, M. Takeda, A Fast String-Searching Algorithm For Multiple Patterns. *Information Processing and Management*, 6 775-791, 1993.

[20] S. Wu, U. Manber. A fast algorithm for multi-pattern searching. Tech. Rep. TR94–17, University of Arizona, Tucson, AZ, 1994.