Faster Algorithms for Computing Maximal Multirepeats in Multiple Sequences

Costas S. Iliopoulos[*]
Algorithm Design Group, Department of Computer Science, King's College London
The Strand, London WC2R 2LS, England, csi@dcs.kcl.ac.uk
Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987, Perth WA 6845, Australia

W. F. Smyth[†*]
Algorithms Research Group, Department of Computing & Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1, smyth@mcmaster.ca
Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987, Perth WA 6845, Australia

Munina Yusufu[†*]
Algorithms Research Group, Department of Computing & Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1, yusufum@mcmaster.ca
Digital Ecosystems & Business Intelligence Institute, Curtin University
GPO Box U1987, Perth WA 6845, Australia

Abstract. A *repeat* in a string is a substring that occurs more than once. A repeat is *extendible* if every occurrence of the repeat has an identical letter either on the left or on the right; otherwise, it is *maximal*. A *multirepeat* is a repeat that occurs at least m_{min} times ($m_{min} \geq 2$) in each of at least $q \geq 1$ strings in a given set of strings. In this paper, we describe a family of efficient algorithms based on suffix arrays to compute maximal multirepeats under various constraints. Our algorithms are faster, more flexible and much more space-efficient than algorithms recently proposed for this

Address for correspondence: Munina Yusufu, Algorithms Research Group, Department of Computing & Software, McMaster University, Hamilton, Ontario, Canada L8S 4K1, yusufum@mcmaster.ca

[*]The author wish to acknowledge the contribution of anonymous referees, whose suggestions materially improved this paper.
[†]The work of the author was supported in part by the Natural Sciences & Engineering Research Council of Canada.

problem. The results extend recent work by two of the authors computing all maximal repeats in a single string.

Keywords: maximal multirepeats, repeats, gaps, biological sequences, suffix arrays

1. Introduction

In this paper, we propose efficient algorithms for finding the maximal multirepeats in a set of strings under various constraints. The problem of finding common regularities among a set of strings is very important [4]. In biological sequences (DNA, RNA, or protein) the problem of locating repeats in a set of strings (multirepeats) arises in many contexts, such as database searching and sequence alignment [1]. It is also important in data mining [8, 3].

A *repeat* in a string is a substring that occurs more than once. The distance (number of intermediate letters) between the occurrences of the same substring is called a *gap*. A repeat is *left-maximal*, if not all occurrences have the same letter on the left, *right-maximal*, if not all occurrences have the same letter on the right — thus *maximal* if both left- and right-maximal. Reporting only maximal repeats avoids redundant reporting of repeats that are embedded in other repeats.

In [10], several fast algorithms for computing different kinds of maximal repeats under some restrictions were proposed, but only for a single string. To compute repeats in a set of strings (*multiple repeats*), there exists only one algorithm [1]. This algorithm is not space-efficient since it uses suffix trees, one for each string in the set plus a "generalized" suffix tree for all of them. Thus it is not easy to implement. In addition, it has high time complexity. If gaps are unrestricted, the algorithm of [1] requires $O(\sigma N^2 n+\alpha)$ time; if gaps are required to fall in a range of length c, it requires $O\big((c^2+\sigma^2)mN^2 n\log(Nn)+\alpha\big)$ time. Here σ is the alphabet size, N the number of strings, n the average length of the N strings, m the *multiplicity* (number of occurrences) of the multirepeat, and α the total number of occurrences of all reported repeats. While n may be quite large (millions), in applications N is generally a small integer (at most two digits). Similarly, we may suppose that the number R of reported repeats is $o(n)$. Further, in keeping with the application, we suppose throughout that alphabet size $\sigma \leq 256$, so that an individual letter requires at most one byte for storage.

Here we extend previous work [10] to the problems considered in [1], proposing algorithms that are more time-efficient, as well as being easier to implement and using much less space. We describe algorithms to find complete maximal multirepeats that occur at least m_{min} times in each of at least q strings in a given set S of N strings, first with no restriction on gap length, then with bounded gaps. For the first problem, we propose two algorithms with worst-case time complexities $O(Nn+\alpha\log_2 N)$ and $O(Nn+\alpha)$ that use $9Nn$ and $10Nn$ bytes of space, respectively. For the second problem, we describe an algorithm with worst-case time complexity $O(RNn)$ that requires approximately $10Nn$ bytes. Note that all times are independent of alphabet size. Extending the algorithms of [1], our three algorithms output only repeats whose occurrences are substrings of length at least p_{min} (user-specified), thus eliminating trivial outputs.

The remainder of the paper is organized as follows. In Section 2, we give definitions and formulate the problems. In Section 3, we give details of the three algorithms noted above. Finally, in Section 4 we give conclusions and thoughts on further research.

2. Preliminaries

Basic string terminology in this paper follows [12].

2.1. Repeats & Data Structures

A *repeat* in x is a tuple $M_{x,u} = (p; i_1, i_2, ..., i_m)$, where $m \geq 2, 1 \leq i_1 < i_2 < ... < i_m \leq n$, and $u = x[i_1..i_1{+}p{-}1] = x[i_2..i_2{+}p{-}1] = ... = x[i_m..i_m{+}p{-}1]$. We call u the *generator*, p the *period* and m the *multiplicity* of $M_{x,u}$. If u occurs at two positions i and j in x, then the distance $g = |i{-}j|{-}p$ is called a *gap*. Note that g may be negative (*overlapping* occurrences) or zero (*tandem* occurrences).

Our second problem considers restrictions on the gaps as follows: if for $i \in 1..\mu{-}1$, where $\mu = m_{min}$, g_i is the gap between the ith and $(i+1)$th occurrences of u, then we require $d_{min_i} \leq g_i \leq d_{max_i}$, lower and upper bounds on g_i. Collectively, these restrictions are represented by a $(\mu{-}1)$-tuple

$$d = \big((d_{min_1}, d_{max_1}), (d_{min_2}, d_{max_2}), \ldots, (d_{min_{\mu-1}}, d_{max_{\mu-1}})\big). \quad (1)$$

As remarked above, our PSY1 algorithm [10] outputs maximal repeats of period $p \geq p_{min}$. For this, certain well-known data structures are required.

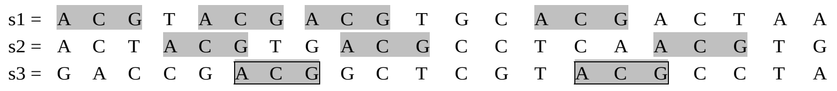
Given a string $s = s[1..\ell]$ of length ℓ, the array $sa = sa[1..\ell]$ is a *suffix array* of s iff its entries are a permutation of $1..\ell$ such that for $j \in 1..\ell$, $sa[j] = i$ whenever suffix $s[i..\ell]$ is the j^{th} in lexicographical order among all the suffixes of s. For brevity, we sometimes refer to $s = s[i..\ell]$ simply as *suffix i*. Often the suffix array is used in combination with the *longest common prefix (lcp)* array which gives the length of the longest common prefix between consecutive suffixes of sa; that is, *lcp*$[j]$ is the length of the longest common prefix of $s[sa[j]..n]$ and $s[sa[j-1]..n]$. Also required is the Burrows-Wheeler transform [2], an array $bwt = bwt[1..\ell]$, most simply defined as follows: for $sa[j] > 1$, $bwt[j] = s\big[sa[j]{-}1\big]$, while for j such that $sa[j] = 1$, $bwt[j] = \$$, a sentinel letter less than any other letter in the alphabet.

2.2. Formulation of Problems

We define two problems:

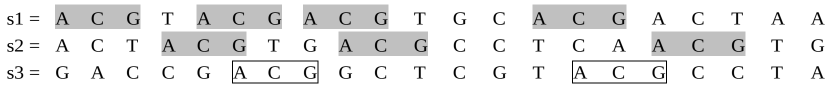
Unconstrained Multirepeats (abbreviated *MultiRep*): Given a set S = $\{s_1, s_2, ..., s_N\}$ of strings, where each string s_k, $1 \leq k \leq N$, has length n (if the lengths of the strings vary, n represents their average length), and a tuple of positive integers $D = (p_{min}, q, m_{min})$, where $p_{min} \geq 1$, $q \in 1..N$, $m_{min} \geq 2$, we output all maximal multirepeats of period at least p_{min} that occur at least m_{min} times in each of at least q strings of S. Following [1], we call q the *quorum* and m_{min} the *minimum multiplicity*.

Example 1: Given a set of three strings S =$\{s_1, s_2, s_3\}$, with $D = (3, 2, 2)$, we find a maximal repeat ACG of length $p_{min} = 3$ that occurs at least $m_{min} = 2$ times in all $3 \geq q = 2$ of the strings. Thus the repeat would be output. However, for $D = (3, 3, 3)$, s_3 would not satisfy $m \geq 3$ and so only $2 < q = 3$ of the strings would have the minimum number of occurrences; in this case no output would occur.


s1 = A C G T A C G A C G T G C A C G A C T A A
s2 = A C T A C G T G A C G C C T C A A C G T G
s3 = G A C C G A C G G C T C G T A C G C C T A

Multirepeats with Constrained Gaps (abbreviated *MultiRepG*): In addition to S and D, we are given a tuple of gap constraints (1). We compute all the repeats that satisfy (1) at least q times as well as the constraints D. More precisely, in each individual string $s_k \in S$ that contains $m \geq m_{min}$ occurrences of the repeating substring, we look for a sequence of $\mu = m_{min}$ consecutive occurrences that satisfies (1); if such a sequence exists in at least q strings, we output all m occurrences in every s_k for which (1) is satisfied.

Example 2: Given the same set S and $D = (3, 2, 2)$ as in Example 1, we introduce the constraint $(d_{min_i}, d_{max_i}) = (0, 5)$ for every $i \in 1..\mu-1$. Because the gap between $s_3[6..8]$ and $s_3[15..17]$ exceeds 5, ACG does not satisfy the gap constraints in s_3, but continues to do so in s_1 and s_2, thus at least $q = 2$ times. Thus occurrences of ACG only in s_1 and s_2 are output.


s1 = A C G T A C G A C G T G C A C G A C T A A
s2 = A C T A C G T G A C G C C T C A A C G T G
s3 = G A C C G A C G G C T C G T A C G C C T A

3. Description of the Algorithms

The overall strategy for both problems MultiRep and MultiRepG is the same:

* form a single string s from the given set S of N strings;
* in a preprocessing phase, compute the suffix array sa, the longest common prefix array lcp and the Burrows-Wheeler transform bwt for s;
* use Algorithm PSY1 [10] to compute all maximal repeats of period $p \geq p_{min}$ in s;
* output the repeats that satisfy D (MultiRep) or both D and d (MultiRepG).

3.1. No Constraints on Gaps

3.1.1. Algorithm MultiRep-1

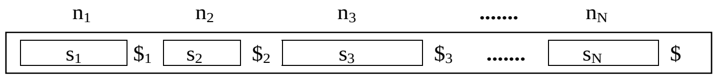


Figure 1. Form a new string using end-of-string sentinels

From the set $S = \{s_1, s_2, \ldots, s_N\}$ of strings, form $s = s_1\$_1 s_2\$_2 s_3\$_3 ... \$_{N-1} s_N\$$, as shown in Figure 1, where the end-of-string sentinels $\$_j$, $1 \leq j \leq N-1$, and \$ are distinct symbols less in lexicographic order than any of the letters in the s_k, $1 \leq k \leq N$, and that moreover satisfy $\$ < \$_1 < \$_2 < ... < \$_{N-1}$. Let $s_k = s_k[1..n_k]$.

The preprocessing computes the sa, lcp and bwt arrays for s using standard algorithms as described in [10]: in these algorithms the $\$_j$ are treated as normal letters, while \$ just marks the end of s and is not included in calculations.

Example 3: Given $S = \{s_1, s_2, s_3\}$, where s_1 = AAGTCAG, s_2 = AGAG, s_3 = CAGTAGC, we form $s = s_1\$_1 s_2\$_2 s_3\$$ and preprocess.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
s	A	A	G	T	C	A	G	$\$_1$	A	G	A	G	$\$_2$	C	A	G	T	A	G	C	\$
sa	8	13	1	6	11	9	18	15	2	20	5	14	7	12	10	19	16	3	17	4	
lcp	-1	0	0	1	2	2	2	2	3	0	1	3	0	1	1	1	1	2	0	1	-1
bwt	G	G	\$	C	G	$\$_1$	T	C	A	G	T	$\$_2$	A	A	A	A	A	A	G	G	\$

PSY1 makes use of the preprocessed arrays to compute maximal repeats, each one a triple $(p; i, j)$ specifying a period $p \geq p_{min}$ and a range $i..j$ in sa such that for every $h \in i..j$, suffix $sa[h]$ has an identical prefix of length p, while suffixes $sa[i-1]$ and $sa[j+1]$ (if they exist) do not. If we are given $p_{min} = 2$ in Example 3, PSY1 would output only one maximal repeat for $\boldsymbol{u} = AG$ in the form $(p; i, j) = (2; 4, 9)$ with period $p = 2$, where the range $4..9$ identifies $sa[4] = 6, sa[5] = 11, sa[6] = 9, sa[7] = 18, sa[8] = 15, sa[9] = 2$. Thus the maximal repeat occurs in positions $6, 11, 9, 18, 15, 2$ of s as shown by the shading in Example 3.

Given an output $(p; i, j)$ from PSY1, we need to determine if the conditions specified by the tuple D are satisfied. Our first task is to use the suffix array sa to convert this output into the form $M = \big(p; sa[i], sa[i+1], \ldots, sa[j]\big)$ keyed to positions in s rather than sa: over all repeats found by PSY1, this will require $O(\alpha)$ time. We then make use of two arrays, *divpts* and *count*. Array $divpts$ specifies the starting points of each substring s_k of s — this permits a binary search to be done to determine in which substring s_k the current repeating substring is located. More precisely:

$$divpts[1..N+1] = [1, n_1+2, n_1+n_2+3, \ldots, \sum_{k=1}^{N-1}(n_k+1)+1, \sum_{k=1}^{N}(n_k+1)+1].$$

The array $count = count[1..N]$ just maintains a count of the number of repeating substrings that have so far been found to lie within each of the N strings s_k.

Using these arrays, it is straightforward to determine in time $O\big((j-i)\log N\big)$ whether the repeat $(p; i, j)$ occurs at least m_{min} times in each of at least q substrings of s, as shown in Figure 2. Note that if $j-i+1 < m_{min}q$, this condition cannot be satisfied and so no tests are required. The function BinarySearch called in MultiRep-1 (see Figure 2) returns the index k indicating that position $sa[h]$ in s occurs in substring s_k.

In Example 3, $divpts$ will be $[1, 9, 14, 22]$ and the output repeat will be $(2; 6, 11, 9, 18, 15, 2)$. After binary search we find that $count[1] = 2$ (positions 6 and 2), $count[2] = 2$ (11 and 9), and $count[3] = 2$ (18 and 15): the repeat occurs at least twice in each of the three substrings. Thus for $m_{min} = 2, q = 3$, the repeat satisfies the constraints specified by D.

Now we analyze the time and space complexity of the algorithm. For construction of sa there are algorithms linear in string length ℓ [7, 5], though in practice algorithms with worst-case $O(\ell^2 \log \ell)$ time requirement are several times faster [11]. To compute lcp from sa there are two linear-time algorithms

```
Input: a maximal multirepeat $M = \big(p; sa[i], sa[i+1], \ldots, sa[j]\big)$ of $s$,
       together with integers $m_{min} \geq 2$, $q \geq 1$.
Output: $M$ if and only if its repeating substring occurs
        at least $m_{min}$ times in each of at least $q$ substrings of $s$.
 — Preprocessing: compute $divpts[1..N{+}1]$.
$r \leftarrow j - i + 1$
if $r \geq qm_{min}$ then
    $count[1..N] \leftarrow 0^N$; $qtotal \leftarrow 0$
    for $h \leftarrow i$ to $j$
        $k \leftarrow$ BinarySearch$(divpts, sa[h])$
        $count[k] \leftarrow count[k] + 1$
        if $count[k] = m_{min}$ then
            $qtotal \leftarrow qtotal + 1$
    if $qtotal \geq q$ then
        output$(M)$
```

Figure 2. Algorithm MultiRep-1: Check Multiplicity & Quorum

[6, 9], and the easy calculation of bwt from sa is also linear. Given lcp and bwt, PSY1 executes in linear time [10]. In our case $\ell = N(n{+}1)$, and so all the repeats $(p; i, j)$ in s can be computed in time $O(Nn)$. For each of $O(R)$ repeats, the array $count$ must be cleared at a cost of $O(N)$ time. In addition, for each of at most α occurrences of repeating substrings in s, the time required is at most $O(\log_2 N)$ for the binary search. Thus to compute all the repeats satisfying constraint D, the worst-case time complexity of the algorithm shown in Figure 2 is $O(Nn{+}RN{+}\alpha \log_2 N)$.

However, the asymptotic time complexity of MultiRep-1, though not perhaps the expected running time in practice, can be slightly reduced, as we now explain. Instead of performing $count \leftarrow 0^N$ as part of the algorithm, execute it only once as preprocessing over all invocations of MultiRep-1. Introduce into MultiRep-1 a list L, initially empty, to which each value k computed by BinarySearch is added; then at the end of MultiRep-1 introduce a new loop that removes from L each entry k and performs $count[k] \leftarrow 0$. The resulting algorithm executes in time $O(Nn{+}\alpha \log_2 N)$, independent of R.

Preprocessing for a string of length ℓ requires as few as 5ℓ bytes for sa [11], 9ℓ for lcp [9] and 6ℓ for bwt, thus at most $9N(n{+}1)$ bytes for $\ell = N(n{+}1)$. PSY1 itself requires only 5ℓ bytes for its execution [10], plus a further 4ℓ for storage of sa (since each range i, j in sa needs to be converted into a sequence $sa[i], sa[i+1], \ldots, sa[j]$ in s). Since $divpts$ and $count$ are arrays $1..N$ of integer, their total space requirement is $8N$ bytes, and so the total is $N(9n{+}17)$ bytes, in simple terms $9Nn$.

The algorithm shown in Figure 2 outputs all of the repeats M. It may instead be required to output only those positions in M that occur in the s_k for which $m \geq m_{min}$. One way to accomplish this is to introduce a Boolean array $mok = mok[1..N]$ (similar in its role to the array $gapsok$ described below for MultiRepG) — mok records for each $k \in 1..N$ whether or not s_k contains at least m_{min} occurrences of M. Then a straightforward processing of M, again using BinarySearch, produces the required output, using the same asymptotic time and space.

3.1.2. Algorithm MultiRep-2

We briefly describe a strategy to avoid the binary search of MultiRep-1, at a cost of an additional $N(n{+}1)$ bytes of storage (based on the assumption that N is small — less than 256). In the preprocessing stage we introduce an array pos of byte such that, for each $i \in 1..N(n+1)$, $pos[i] = k$ iff i is a position in s_k, while otherwise $pos[i] = 0$ ($s[i]$ is a sentinel). Thus for every i, $pos[i] \in 0..N$. Using $divpts$, pos can easily be computed in $\Theta(Nn)$ time. Then, in order to determine, for each position h in sa which substring s_k the position $sa[h]$ occurs in, it is necessary only to compute $k \leftarrow pos\big[sa[h]\big]$. This $O(1)$ computation replaces BinarySearch in MultiRep-1, reducing processing time to $O(Nn{+}\alpha)$, thus asymptotically optimal.

3.2. Restricted Gaps (MultiRepG)

In this section, we introduce the algorithm MultiRepG, for which the input is a maximal multirepeat $(p; i, j)$ of s satisfying constraints $D = (p_{min}, q, m_{min})$ and the output consists of the elements of $(p; i, j)$ that satisfy the gap constraints d in at least q substrings s_k of s.

In order to satisfy constraints d in addition to those specified by D, we need to introduce a bit vector $loc = loc[1..N(n+1)]$. In a single preprocessing stage every position in loc is set `FALSE` in time $O(Nn/w)$, where w is the computer word length, and the precondition $loc[h] = \texttt{FALSE}$ for all h is maintained thereafter. Then for *each* maximal repeat $(p; i, j)$ of period $p \geq p_{min}$, the positions $loc\big[sa[h]\big], h = i, i{+}1, \ldots, j$, are set `TRUE`, so that a left-to-right scan of loc will yield in increasing order the positions of the repeating substrings in s. Such a scan is shown in Figure 3, used to determine which of the substrings s_k in s satisfy the gap constraints. A Boolean array $gapsok = gapsok[1..N]$ is used to record the values $k \in 1..N$ for which s_k satisfies d (see the corresponding array mok described earlier for MultiRep-1). Algorithm MultiRepG executes in two phases, a checking phase and an output phase.

In the checking phase, $divpts$ is used to compute for each s_k an array occ of candidate positions. The function $check$, described below, actually applies the constraints d to occ — its total time usage over all invocations is $O(r)$, where $r = j{-}i{+}1 < Nn$; also, the positions inspected in $divpts$ and $gapsok$ for each repeat are at most N. Thus for each candidate repeat, the time required to evaluate the constraints d is $O(Nn)$. For R such repeats, the overall time requirement of the checking phase is therefore $O(RNn)$. We note that since $\alpha \leq RNn$ (the total number α of occurrences of repeats cannot exceed the number R of repeats times the overall string length Nn), therefore $O(RNn)$ in fact represents the total time required both for MultiRep-2 and the checking phase. For cases that arise in practice, a corresponding statement holds also for MultiRep-1.

In the output phase, there is no action if less than q substrings of s contain repeats satisfying the constraints d. Otherwise, occ is recomputed for each s_k that satisfies d and the repeat is then output. For the strings and gap constraints of Example 2, described above, the output of the algorithm given in Figure 3 would be $(p, k, occ) = (3, 1; 1, 5, 8, 14)$ and $(3, 2; 4, 9, 17)$. The overall time requirement of the output phase is again $O(RNn)$.

The Boolean function $check$, shown in Figure 4, slides a window of width m_{min} over the $m \geq m_{min}$ entries in occ, corresponding to the substring s_k, shifting right by one position at each step. For each window, $check$ determines whether its entries satisfy the constraints d; if so, $check$ returns `TRUE`, causing the m repeating substrings of occ that occur in s_k to be output. If no window of occ satisfies

```
 — Precondition: $loc = \texttt{FALSE}^{N(n+1)}$.
for $h \leftarrow i$ to $j$ do $loc\big[sa[h]\big] \leftarrow \texttt{TRUE}$
 — First Phase: Checking
$q' \leftarrow 0;\ gapsok[1..N] \leftarrow \texttt{FALSE}^N$
$k \leftarrow 1;\ m \leftarrow 0; r \leftarrow j-i+1;\ r' \leftarrow 0;\ h \leftarrow 1$
while $r' < r$ do
    if $loc[h]$ then
        $r' \leftarrow r'+1$
        if $h < divpts[k+1]$ then $m \leftarrow m+1;\ occ[m] \leftarrow h$
        else
            if $m \geq m_{min}$ and $check(p, occ, m, d, m_{min})$ then
                $q' \leftarrow q'+1;\ gapsok[k] \leftarrow \texttt{TRUE}$
            $m \leftarrow 1;\ occ[1] \leftarrow h$
            repeat $k \leftarrow k+1$ until $h < divpts[k+1]$
    $h \leftarrow h+1$
if $m \geq m_{min}$ and $check(p, occ, m, d, m_{min})$ then
    $q' \leftarrow q'+1;\ gapsok[k] \leftarrow \texttt{TRUE}$
 — Second Phase: Output
if $q' \geq q$ then
    for $k \leftarrow 1$ to $N$ do
        if $gapsok[k]$ then
            $m \leftarrow 0$
            for $h \leftarrow divpts[k]$ to $divpts[k+1]-1$ do
                $m \leftarrow m+1;\ occ[m] \leftarrow h$
            output$(p, k, occ)$
for $h \leftarrow i$ to $j$ do $loc\big[sa[h]\big] \leftarrow \texttt{FALSE}$
```

Figure 3. Algorithm MultiRepG: for each substring s_k of s, if occ contains a sequence of length $\mu = m_{min}$ that satisfies (1), then output occ

d, $check$ returns FALSE. The constraints d are accessed as a two-dimensional array $d[1..m_{min}-1, 1..2]$. The outer **while** loop of $check$ is executed $(m - m_{min} + 1)$ times in the worst case, and the inner **while** loop is executed at most m_{min} times; thus the execution time of $check$ at each invocation is $O(m_{min}(m - m_{min} + 1)) = O(m)$. Here we assume that the specified input value m_{min} is constant over the execution of the algorithm. Over all invocations, therefore, the execution time of $check$ is $O(r)$.

We note that the corresponding algorithm described in [1] requires that the differences between the maximum and minimum gaps specified in (1) should all be bounded by a small constant c. The methodology described here requires no such bound, and its effectiveness does not depend on such differences. Note also that MultiRepG can easily be modified, with the same asymptotic complexity and usage of space, to output only those ranges of occ that satisfy d, omitting those entries that do not.

The additional storage required for MultiRepG consists of the $4Nn/w$ bytes for loc plus up to $4n$ bytes for the integer array occ, a total of $4n(N/w+1)$. For $w = 32$, this amounts to $n(N/8+4)$, perhaps as much as an additional Nn bytes on top of the $9Nn$ used by MultiRep-1.

```
function $check(p, occ, m, d, m_{min})$ : boolean
    $I_0 \leftarrow I \leftarrow 1$
    while $m - I \geq m_{min} - 1$ do
        $J \leftarrow 1$
        while $J < m_{min}$ and $d[J,1] \leq occ[I+1] - occ[I] - p \leq d[J,2]$ do
            $I \leftarrow I + 1$; $J \leftarrow J + 1$
        if $J = m_{min}$ then
            return TRUE
        else
            $I_0 \leftarrow I \leftarrow I_0 + 1$
    return FALSE
```

Figure 4. Function $check$: given an array occ of m occurrences of a repeating substring in s_k, determine whether occ contains a subarray of length $\mu = m_{min}$ that satisfies the constraints d

Table 1 compares the algorithms described here with those proposed in [1]. Note that even though

Problem	Algorithm	Time	Space
MultiRep	[1]	$O(\sigma N^2 n + \alpha)$	linear but large
	MultiRep-1	$O(Nn + \alpha \log_2 N)$	$9Nn$ bytes
	MultiRep-2	$O(Nn + \alpha)$	$10Nn$ bytes
MultiRepG	[1]	$O((c^2 + \sigma^2) m N^2 n \log(Nn)) + \alpha)$	$O(c^2 Nnm)$
	MultiRepG	$O(RNn)$	$10Nn$ bytes

Table 1. Comparison of Algorithms.

the suffix tree storage is linear, the large amount of information in each edge and node makes the suffix tree very expensive, consuming about ten to twenty times the memory size of the input text in good implementations. In [1], the algorithm MultiRep uses suffix trees, one for each string in the set plus a "generalized" suffix tree for all of them, therefore the memory usage would be very large.

4. Discussion

We have formulated two problems related to multirepeats in sets of strings with various restrictions and presented efficient algorithms with lower time complexity and less memory consumption compared to previously proposed algorithms. We remark that if in Algorithm MultiRepG we set the min and max constraints on gaps equal to zero, we can find all tandem repeats (repetitions) in arbitrary subsets of S. Future work includes the detection of degenerate (approximate) multirepeats and weighted multirepeats.

References

[1] A. Bakalis, Costas S. Iliopoulos, Christos Makris, Spyros Sioutas, Evangelos Theodoridis, Athanasios K. Tsakalidis, Kostas Tsichlas: Locating maximal multirepeats in multiple strings under various constraints, *The Computer Journal 50–2*, 2007, 178–185.

[2] Michael Burrows, David J. Wheeler: *A Block-Sorting Lossless Data Compression Algorithm*, Technical Report 124, Digital Equipment Corporation, 1994.

[3] Johannes Fischer, Volker Heun, Stefan Kramer: Optimal string mining under frequency constraints, *Proc. 10th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, LNCS 4213, Springer-Verlag, 2006, 139–150.

[4] Dan Gusfield: *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997.

[5] Juha Kärkkäinen, Peter Sanders: Simple linear work suffix array construction, *Proc. 30th Internat. Colloq. Automata, Languages & Programming*, LNCS 2971, Springer-Verlag, 2003, 943–955.

[6] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, Kunsoo Park: Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proc. 12th Annual Symp. Combinatorial Pattern Matching*, LNCS 2089, Springer-Verlag, 2001, 181–192.

[7] Pang Ko, Srinivas Aluru: Space efficient linear time construction of suffix arrays, *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, R. Baeza-Yates, E. Chávez & M. Crochemore (eds.), LNCS 2676, Springer-Verlag, 2003, 200–210.

[8] Sau Dan Lee, Luc De Raedt: An efficient algorithm for mining string databases under constraints, *Proc. KDID*, LNCS 3377, Springer-Verlag, 2005, 108–129.

[9] Giovanni Manzini: Two space saving tricks for linear time LCP computation, *Proc. 9th Scandinavian Workshop on Algorithm Theory*, LNCS 3111, Springer-Verlag, 2004, 372–383.

[10] Simon J. Puglisi, W. F. Smyth, Munina Yusufu: Fast optimal algorithms for computing all the repeats in a string, *Prague Stringology Conference*, Jan Holub & Jan Žd’árek (eds.), 2008, 161–169.

[11] Simon J. Puglisi, W. F. Smyth, Andrew Turpin: A taxonomy of suffix array construction algorithms, *ACM Computing Surveys 39–2*, Article 4, 2007.

[12] Bill Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, 2003.