Crochemore's repetitions algorithm revisited – computing runs

Frantisek Franek* and Mei Jiang

Department of Computing & Software
Faculty of Engineering
McMaster University
Hamilton, Ontario
Canada L8S 4K1
franek@mcmaster.ca jiangm5@mcmaster.ca

Abstract. Crochemore's repetitions algorithm introduced in 1981 was the first $O(n \log n)$ algorithm for computing repetitions. Since then, several linear-time worst-case algorithms for computing runs have been introduced. They all follow a similar strategy: first compute the suffix tree or array, then use the suffix tree or array to compute the Lempel-Ziv factorization, then using the Lempel-Ziv factorization compute all the runs. It is conceivable that in practice an extension of Crochemore's repetitions algorithm may outperform the linear-time algorithms, or at least for certain classes of strings. The nature of Crochemore's algorithm lends itself naturally to parallelization, while the linear-time algorithms are not easily conducive to parallelization. For all these reasons it is interesting to explore ways to extend the original Crochemore's repetitions algorithm to compute runs. We present three variants of extending the repetitions algorithm to compute runs: two with a worsen complexity of $O(n(\log n)^2)$, and one with the same complexity as the original algorithm. The three variants are tested for speed of performance and their memory requirements are analyzed. The third variant is tested and analyzed for various memory-saving alterations. The purpose of this research is to identify the best extension of Crochemore's algorithm for further study, comparison with other algorithms, and parallel implementation.

Keywords: repetition, run, string, periodicity, suffix tree, suffix array

1 Introduction

An important structural characteristic of a string over an alphabet is its periodicity. Repetitions (tandem repeats) have always been in the focus of the research into periodicities. The concept of runs that captures maximal fractional repetitions which themselves are not repetitions was introduced by Main [12] as a more succinct notion in comparison to repetitions. The term run was coined by Iliopoulos et al. [8]. It was shown by Crochemore in 1981 that there could be $O(n \log n)$ repetitions in a string of length n and an $O(n \log n)$ time worst-case algorithm was presented [3] (a variant is also described in Chapter 9 of [4]), while Kolpakov and Kucherov proved in 2000 that the number of runs was $O(n)$ [9].

Since then, several linear-time worst-case algorithms have been introduced, all based on linear algorithms for computing suffix trees or suffix arrays. Main [12] showed how to compute the leftmost occurrences of runs from the Lempel-Ziv factorization in linear time, Weiner [14] showed how to compute Lempel-Ziv factorization from a suffix tree in linear time. Finally, in 1997 Farach [6] demonstrated a linear construction

* Supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada.

Frantisek Franek, Mei Jiang: Crochemore's repetitions algorithm revisited – computing runs, pp. 214–224.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

of suffix tree. In 2000, Kolpakov and Kucherov [9] showed how to compute all the runs from the leftmost occurrences in linear time. Suffix trees are complicated data structures and Farach construction was not practical to implement for sufficiently large n, so such a linear algorithm for computing runs was more of a theoretical result than a practical algorithm.

In 1993, Manber and Myers [13] introduced suffix arrays as a simpler data structure than suffix trees, but with many similar capabilities. Since then, many researchers showed how to use suffix arrays for most of the tasks suffix trees were used without worsening the time complexity. In 2004, Abouelhoda et al. [1] showed how to compute in linear time the Lempel-Ziv factorization from the extended suffix array. In 2003, several linear time algorithms for computing suffix arrays were introduced (e.g. [10,11]). This paved the way for practical linear-time algorithms to compute runs. Currently,there are several implementations(e.g. Johannes Fischer's, Universität Tübingen, or Kucherov's, CNRS Lille) and the latest, CPS, is described and analyzed in [2].

Though suffix arrays are much simpler data structures than suffix trees, these linear time algorithms for computing runs are rather involved and complex. In comparison, Crochemore's algorithm is simpler and mathematically elegant. It is thus natural to compare their performances. The strategy of Crochemore's algorithm relies on repeated reffinements of classes of equivalence, a process that can be easily parallelized, as each reffinement of a class is independent of the other classes and their reffinements, and so can be performed simultaneously by different processors. The linear algorithms for computing runs are on the other hand not very conducive to parallelization (the major reason is that all linear suffix array constructions rely on recursion). For these reasons we decided to extend the original Crochemore's algorithm based on the most memory efficient implementation by Franek et.al. [4]. In this report we discuss and analyze three possible extensions of [4] for computing runs and their performance testing: two variants with time-complexity of $O(n(\log n)^2)$ and one variant with time-complexity of $O(n \log n)$. Two diffierent menthods to save memory for the third variant are tested and analyzed. The purpose of this study was to identify the best extension of Crochemore's repetitions algorithm to compute runs for comaprison with other runs algorithm and for parallel implementation.

2 Basic notions

Repeat is a collection of repeating substrings of a given string. *Repetition*, or *tandem repeat*, consists of two or more adjacent identical substrings. It is natural to code repetitions as a triple (s, p, e), where s is the *start* or *starting position* of the repetition, p is its *period*, i.e. the length of the repeating substring, and e is its *exponent* (or *power*) indicating how many times the repeating substring is repeated. The repeating substring is referred to as the *generator* of the repetition. More precisely:

Definition 1. (s, p, e) *is a repetition in a string* $x[0..n-1]$ *if*
$x[s..(s+p-1)] = x[(s+p)..(s+2p-1)] = \cdots = x[(s+(e-1)p)..(s+ep-1)]$.
A repetition (s, p, e) *is* maximal *if it cannot be extended to the left nor to the right, i.e.* (s, p, e) *is a repetition in* x *and* $x[(s-p+1)..(s-1)] = x[s..(s+p-1)]$ *and*
$x[(s+(e-1)p)..(s+ep-1)] = x[(s+ep)..(s+(e+1)p-1)]$.

In order to make the coding of repetitions more space efficient, the repetitions with generators that are themselves repetitions are not listed; for instance, `aaaa` should be

reported as (0,1,4) just once, there is no need to report (1,2,2) as it is subsumed in (0,1,4).
Thus we require that generator of a repetition be *irreducible*, i.e. not a repetition.

Consider a string `abababa`, there are maximal repetitions (0,2,3) and (1,2,3). But, in fact, it can be viewed as a fractional repetition (0,2,3+$\frac{1}{2}$). This is an idea of a run coded into a quadruple (s, p, e, t), where s, p, and e are the same as for repetitions, while t is the *tail* indicating the length of the last incomplete repeat. For instance, for the above string we can only report one run (0,2,3,1) and it characterizes all the repetitions implicitly. The notion of runs is thus more succinct and more space efficient in comparison with the notion of repetitions. More precisely:

Definition 2. *$x[s..(s+ep+t)]$ is a* run *in a string $x[0..n-1]$ if $x[s..(s+p-1)] = x[(s+p)..(s+2p-1)] = \cdots = x[(s+(e-1)p)..(s+ep-1)]$ and $x[(s+(e-1)p)..(s+(e-1)p+t)] = x[(s+ep)..(s+ep+t)]$, where $0 \leq s < n$ is the* start *or the* starting position *of the run, $1 \leq p < n$ is the* period *of the run, $e \geq 2$ is the* exponent *(or* power*) of the run, and $0 \leq t < p$ is the* tail *of the run. Moreover, it is required that either $s = 0$ or that $x[s-1] \neq x[s+2p-1]$ (in simple terms it means that it cannot be extended to the left) and that $x[s+(ep)+t+1] \neq x[s+(e+1)p+t+1]$ (in simple terms it means that the tail cannot be extended to the right). It is also required, that the generator be irreducible.*

3 A brief description of Crochemore's algorithm

Let $x[0..n-1]$ be a string. We deffine an equivalence $\approx_p$ on positions $\{0, \ldots, n-1\}$ by $i \approx_p j$ if $x[i..i+p-1] = x[j..j+p-1]$. In Fig. 1, the classes of $\approx_p$, $p = 1..8$ are illustrated. For technical reasons, a sentinel symbol \$ is used to denote the end of the input string; it is considered to be the lexicographically smallest character. If $i, i+p$ are in the same class of $\approx_p$ (as illustrated by 5,8 in the class $\{0, 3, 5, 8, 11\}$ on level 3, or 0,5 in class $\{0, 5, 8\}$ on level 5, in Fig. 1) then there is a tandem repeat of period p (thus $x[5..7] = x[8..10] =$`aba` and $x[0..4] = x[5..9] =$`abaab`). Thus the computation of the classes and identiffication of repeats of the same "gap" as the level (period) being computed lay in the heart of Crochemore's algorithm. A naive approach following the scheme of Fig. 1 would lead to an $O(n^2)$ algorithm, as there are potentially $\leq n$ classes on each level and there can be potentially $\leq \frac{n}{2}$ levels.

The first level is computed directly by a simple left-to-right scan of the input string - of course we are assuming that the input alphabet is $\{0, \ldots, n-1\}$, if it is not, in $O(n \log n)$ the alphabet of the input string can be transformed to it.

Each follow-up level is computed from the previous level by refinement of the classes of the previous level (in Fig. 1 indicated by arrows). Once a class decreases to a singleton (as $\{15\}$ on level 1 , or $\{14\}$ on level 2), it is not refined any further. After a level p is computed, the equivalent positions with "gap" are identified, extended to maximum, and reported. Note that the levels do not need to be saved, all we need is a previous level to compute the new level (which will become the previous level in the next round). When all classes reach its final singleton stage, the algorithm terminates.

How to compute next level from the previous level – refinement of a class by class. Consider a refinement of a class $\mathcal{C}$ on level L by a class $\mathcal{D}$ on level L: take $i, j \in \mathcal{C}$, if $i+1, j+1 \in \mathcal{D}$, then we leave them together, otherwise we must separate them. For instance, let us refine a class $\mathcal{C} = \{0, 2, 3, 5, 7, 8, 10, 11, 13\}$ by a class

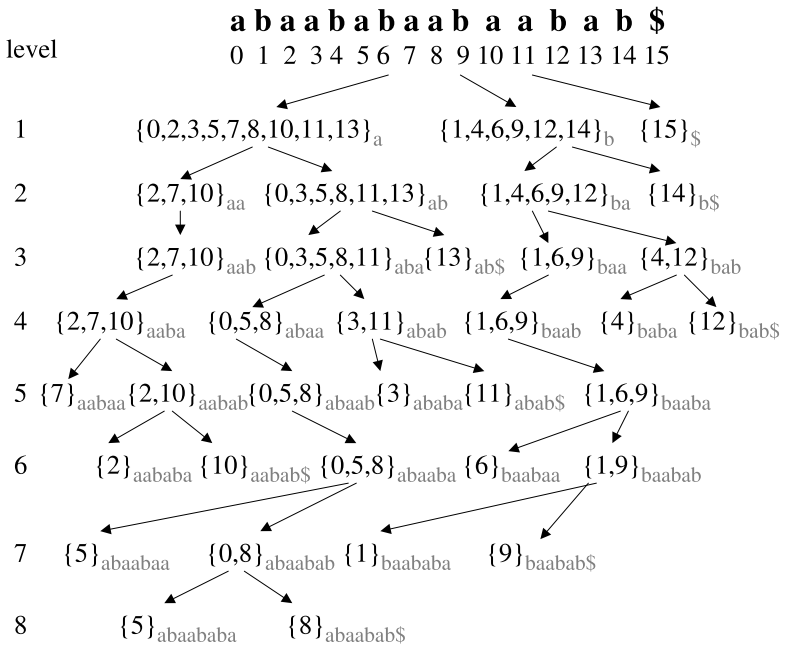

Figure 1. Classes of equivalence and their refinements for a string `abaababaabaabab`

$\mathcal{D} = \{1, 4, 6, 9, 12, 14\}$ on level 1. 0 and 2 must be separated as 1,3 are not both in $\mathcal{D}$, 0 and 3 will be in the same class, since 1,4 are both in $\mathcal{D}$. In fact $\mathcal{C}$ will be refined into two classes, one consisting of $\mathcal{D}$ shifted one position to the left ($\{0, 3, 5, 8, 11, 13\}$), and the ones that were separated ($\{2, 7, 10\}$). If we use all classes for refinement, we end up with the next level.

A major trick is not to use all classes for refinement. For each "family" of classes (classes that were formed as a refinementof a class on the previous level – for instance classes $\{2, 7, 10\}$ and $\{0, 3, 5, 8, 11, 13\}$ on level 2 form a family as they are a refinement of the class $\{0, 2, 3, 5, 7, 8, 10, 11, 13\}$ on level 1). In each family we identify the largest class and call all the others small. By using only small classes for refinement, $O(n \log n)$ complexity is achieved as each element belongs only to $O(\log n)$ small classes.

Many linked lists are needed to be maintained to keep track of classes, families, the largest classes in families, and gaps. Care must be taken to avoid traversing any of these structure lest the $O(n \log n)$ complexity be compromised. It was estimated that an implementation of Crochemore's algorithm requires about $20 * n$ machine words. FSX03 [4] managed to trim it down to $14 * n$ using memory multiplexing and virtualization without sacrificing either the complexity or much of the performance.

4 Extending Crochemore's algorithm to compute runs

One of the features of Crochemore's algorithm is that

(a) repetitions are reported level by level, i.e. all repetitions of the same period are reported together, and
(b) there is no order of repetition reporting with respect to the starting positions of the repetitions (this is a byproduct of the process of refinement),

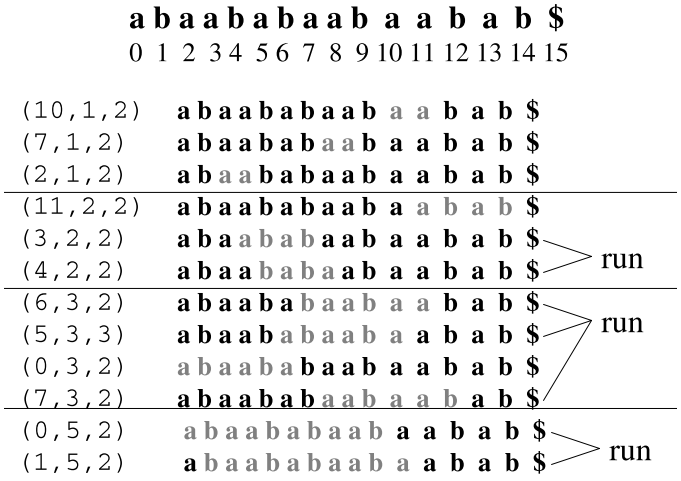

Figure 2. Reporting repetitions for string `abaababaabaabab`

and thus the repetitions must be "collected" and "joined" into runs. For instance, for a string $x =$`abaababaabaabab`, the order of repetitions as reported by the algorithm FSX03 ([4]) is shown in Fig. 2; it also shows some of the repetitions that have to be joined into runs.

The first aspect of Crochemore's algorithm (see (a) above) is good for computing runs, for all candidates of joining must have the same period. The second aspect (see (b) above) is detrimental, for it is needed to check for joining two repetitions with "neigbouring" starts.

4.1 Variant A

In this variant all repetitions for a level are collected, joined into runs, and reported. The high level logic:

1. Collect the runs in a binary search tree based on the starting position. There is no need to record the period, as all the repetitions and all the runs dealt with are of the same period.
2. When a new repetition is reported, find if it should be inserted in the tree as a new run, or if it should be joined with an existing run.
3. When all repetitions of the period had been reported, traverse the tree and report all runs (if depth first traversal is used, the runs will be reported in order of their starting positions).

The rules for joining:

1. Descend the tree as if searching for a place to insert the newly reported repetition.
2. For every run encountered, check if the repetition should be joined with it.
   (a) If the repetition is a substring of the run, ignore the repetition and terminate the search.
   (b) If the run is a substring of the repetition, replace the run with the repetition and terminate the search.
   (c) If the run's starting position is to the left of the starting position of the repetition, if the run and the repetition have an overlap of size $\geq p$, the run's tail

must be updated to accommodate the repetition (i.e. the run is extended to the right). On the other hand, if the overlap is of size $< p$ or empty, continue search.

(d) If the run's starting position is to the right of the starting position of the repetition, if the repetition and the run have an overlap of size $\geq p$, the run's starting position must be updated to accommodate the repetition (i.e. the run is extended to the left). On the other hand, if the overlap is of size $< p$ or empty, continue search.

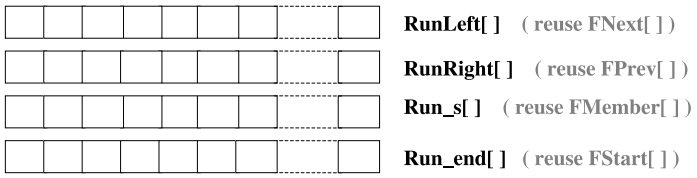

Figure 3. Data structures for Variant A

For technical reasons and to lower memory requirements, the runs are recorded in the search tree as pairs (s, d) where s is the starting position of the run, while d is the end position of the run (let us remark again that we do not need to store the period p). Note that we can easily compute the exponent: $e = (d-s+1)$ / p, and the tail $t = (d-s+1)$ % p.

To avoid dynamic memory allocation and the corresponding deterioration of performance, the search tree is emulated by 4 integer arrays of size n, named `RunLeft[]` (emulating pointers to the left children), `RunRight[]` (emulating pointers to the right children), `Run_s[]` (emulating storing of the starting position in the node), and `Run_d[]` (emulating storing of the endposition in the node), see Fig. 3. Since the four arrays, `FNext[]`, `FPrev[]`, `FMember[]`, and `FStart[]`, are used in the underlying Crochemore's algorithm only for class refinement, and at the time of repetition reporting they can be used safely (as long as they are properly "cleaned" after the use), we do not need any extra memory.

Thus the variant A does not need any extra memory as each search tree is "destroyed" after the runs have been reported, however there is an extra penalty of traversing a branch of the search tree for each repetition reporting, i.e. extra $O(\log n)$ steps, leading to the complexity of $O(n(\log n)^2)$.

4.2 Variant B

In this variant all repetitions for all levels are collected, joined into runs, and reported together at the end.

The basic principles are the same as for variant A. However, for each level we build a separate search tree and keep it till the repetitions of all levels (periods) have been reported. We cannot use any of the data structures from the underlying Crochemore's algorithm as we did for variant A, so the memory requirement grows by additional $4 * n$ machine words. The time-complexity is the same as for variant A, i.e. $O(n(\log n)^2)$.

How do we know that all the runs can fit into the search trees with a total of n nodes? We do not know, for it is just a conjecture that the maximum number of

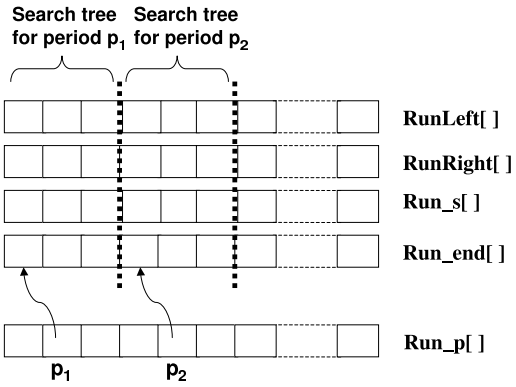

Figure 4. Data structures for Variant B

runs $< n$. However, if we run out of the space (there is a safeguard), we will have found a counterexample to the conjecture on the maximum number of runs (see e.g. [5]).

4.3 Variant C

As in Variant B, all repetitions for all levels are collected, joined into runs, and reported together at the end. However, this variant differs from B in the data structure used.

The repetitions are collected in a simple data structure consisting of an array `Buckets[]`. In the bucket `Buckets[`s`]` we store a simple singly-linked list of all repetitions that start at position s. To avoid as much as possible dynamic allocation, so-called "allocation-from-arena" technique is used for the linked lists (`Buckets[]` is allocated with the other structures) and $3 * n$ words is allocated in chunks as needed. The memory requirement for collecting and storing all the repetitions is $\leq 4n * \log n$ words, however an expected memory requirement is $4n$ words as the expected number of repetitions is n ($3n$ for the links, n for the buckets).

After all repetitions had been reported and collected, `Buckets[]` is traversed from left to right and all repetitions are joined into runs - we call this phase "sweep". In another traversal,the runs can bereported. During the sweep, everything to the left of the current index are runs, while everything to the right and including the current

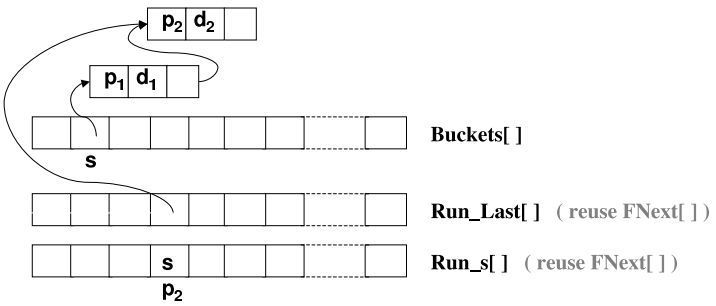

Figure 5. Data structures for Variant C

index are repetitions. For the joining business, we need for each period to remember the rightmost run with that period, that is the role of the array `RunLast[]` (we can reuse `FNext[]`). Thus when traversing the linked list in the bucket `Buckets[`i`]` and currently dealing with a repetition with period p_2, `RunLast[`p_2`]` points to the last run of period p_2 so we can decide if the current repetition is to be "promoted" to a run (with a zero tail), or joined with the run. Since the starting position of the last run of period p_2 is not stored in the run, we need one more array `Run_s[]` in which we store the starting position (we can reuse `FPrev[]`).

Since storing a repetition in `Buckets[]` takes a constant time, and there are $O(n \log n)$ repetitions, and since the joining business is also constant time, the overall time complexity is $O(n \log n) + O(n \log n)$, i.e. $O(n \log n)$.

5 Experimental results

Implementations of the three variants were compared as to their performance. The testing was rather informal, just to give indications how the three variants compare.
Hardware: Sony VAIO laptop with Intel Core-2 Duo CPU T5800 @ 2.00 GHz, 4 GB of RAM
Software: Windows Vista Home Premium SP1. The code was written in C++ and was compiled using the GNU `g++` compiler.
Each run was repeated five times, the minimum numbers are recorded in the table given in Fig. 6 (`random2.txt` is a file of random strings on a binary alphabet, while `random21.txt` is a file of random strings on an alphabet of size 21).

Data Set	File Name	String Length	Time (seconds)		
			Variant A	Variant B	Variant C
DNA	dna.dna4	510976	105.87	110.15	3.12
English	bible.txt	4047392	63.27	62.65	23.90
Fibonacci	fibo.txt	305260	173.30	177.00	2.39
Periodic	fss.txt	304118	159.61	168.78	2.44
Protein	p1Mb.txt	1048576	47.93	53.23	5.15
Protein	p2Mb.txt	2097152	189.20	189.98	11.42
Random	random2.txt	510703	193.01	189.28	4.42
Random	random21.txt	510703	7.69	7.46	1.89

Figure 6. Comparing speed performance of variants A, B, and C

The table given in Fig. 7 records the performance averaged per a character of input:

The results allow for a quick conclusion:

1. Overall, variant C is significantly faster than variants A and B. In fact by 3643%!
2. Even though variant A requires less additional memory, speed-wise does not do much better than B.
3. The speed of variants A and B is not proportional to the string's length. Rather, it mostly depends on the type of the string. It works better on strings with large alphabet size and low periodicity. This is intuitively clear, as for high periodicity strings the height of the search trees are large.

6 Memory-saving modifications of Variant C

In the first modification, C1, repetitions are collected for a round of K levels, then a sweep is executed and the resulting runs are reported, and the bucket memory is then

Data Set	File Name	Name	# of runs	Time (µsec / letter)		
				Variant A	Variant B	Variant C
DNA	dna.dna4	510976	130368	207.18	215.57	6.11
English	bible.txt	4047392	63690	15.63	15.48	5.91
Fibonacci	fibo.txt	305260	233193	567.70	579.83	7.82
Periodic	fss.txt	304118	281912	524.84	554.98	8.01
Protein	p1Mb.txt	1048576	69605	45.71	50.76	4.91
Protein	p2Mb.txt	2097152	139929	90.22	90.59	5.45
Random	random2.txt	510703	210122	377.93	370.62	8.64
Random	random21.txt	510703	24389	15.06	14.60	3.70
Overall average				**230.53**	**236.55**	**6.32**

Figure 7. Comparing speed performance of variants A, B, and C per character of input

reused in the next "batch" of repetitions. For our experiments, we used $K = 100$, so we refer to this variant as C1-100.

In the second modification, C2, we consolidate repetitions with small periods ($\leq K$) into runs when putting them to the buckets (this saves memory since there are fewer runs than repetitions). For a repetition with periond $p \leq K$ and start s, we check p buckets to the left and to the right of s; for $p > K$, we check K buckets to the left and to the right of s. This guarantes that all reptitions up to period K have been consolidated into runs before the final sweep, while repetitions of periods $> K$ are partially consolidated. Thus the final sweep ignores the repetitions with periods $\leq K$. Beside saving memory, the final sweep is a bit shorter, while putting repetitions into the buckets is a bit longer. For our experiemts, we used $K = 10$, so we refer to this variant as C2-10.

The table given in Fig. 8 show comparisons of C, C1-100, and C2-10 for the speed of performance on the same datasets as the tests among the variants A, B, and C in tables in Fig. 6 and Fig. 7.

Data Set	File Name	File size (bytes)	# of runs	Time (seconds)		
				C	C1-100	C2-10
DNA	dna.dna4	510976	130368	3.02	3.04	2.87
English	bible.txt	4047392	63690	20.29	20.36	20.53
Fibonacci	fibo.txt	305260	233193	2.75	6.60	2.76
Periodic	fss.txt	304118	281912	2.65	5.34	3.02
Protein	p1Mb.txt	1048576	69605	4.47	4.52	4.42
Protein	p2Mb.txt	2097152	139929	10.21	10.56	10.29
Random	random2.txt	510703	210122	4.15	4.16	4.01
Random	random21.txt	510703	24389	1.59	1.65	1.57

Figure 8. Comparing speed performance of the variants C, C1-100, and C2-10

As expected, C is the fastest, however the differences are insignificant, except somehow significant results for `fibo.txt` and `fss.txt`.

The table given in Fig. 9 show comparisons of memory usage of C, C1-100, and C2-10.

Only on `fibo.txt` and `fss.txt` C1-100 and C2-10 exhibit memory savings, for all other data sets, the memory requirements are the same corresponding to the string's length (i.e. only 1 arena segment is allocated).

For the next set of tests we used large strings with large number of runs. The strings were obtained from W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and

Data set	File name	File size (bytes)	Alphabet size	# of runs	Memory (blocks)		
					C	C1-100	C2-10
DNA	dna.dna4	510976	5	130368	510976	510976	510976
English	bible.txt	4047392	63	63690	4047392	4047392	4047392
Fibonacci	fibo.txt	305260	2	233193	2747340	1221040	610520
Periodic	fss.txt	304118	2	281912	1824708	912354	608236
Protein	p1Mb.txt	1048576	23	69605	1048576	1048576	1048576
Protein	p2Mb.txt	2097152	23	139929	2097152	2097152	2097152
Random	random2.txt	510703	2	210122	510703	510703	510703
Random	random21.txt	510703	21	24389	510703	510703	510703

Figure 9. Comparing memory usage of the variants C, C1-100, and C2-10

A. Shinohara's website dedicated to "Lower Bounds for the Maximum Number of Runs in a String" at URL `http://www.shino.ecei.tohoku.ac.jp/runs/`.

The table in Fig. 10 indicates the time performance C, C1-100, and C2-10 on these run-rich large strings, while the table in Fig. 11 gives the memory usage.

File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
				C	C1-100	C2-10
60064.txt	60064	2	56714	0.34	0.51	0.34
79568.txt	79568	2	75136	0.50	0.72	0.56
105405.txt	105405	2	99541	0.70	1.05	0.79
139632.txt	139632	2	131869	1.06	1.59	1.10
176583.txt	176583	2	166772	1.71	2.58	1.43
184973.txt	184973	2	174697	1.63	2.78	1.46

Figure 10. Comparing speed of C, C1-100, and C2-10 on large run-rich strings

File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
				C	C1-100	C2-10
60064.txt	60064	2	56714	240256	180192	120128
79568.txt	79568	2	75136	318272	238704	159136
105405.txt	105405	2	99541	527025	316215	210810
139632.txt	139632	2	131869	698160	418896	279264
176583.txt	176583	2	166772	882915	529749	353166
184973.txt	184973	2	174697	924865	554919	369946

Figure 11. Memory usage of C, C1-100, and C2-10 on large run-rich strings

As expected, for strings with many short runs and a few long runs, C2-10 exhibits significant memory savings, with little performance degradation.

7 Conclusion and further research

We extended Crochemore's repetitions algorithm to compute runs. Of the three variants, variant C is by far more efficient time-wise, but requiring $O(n \log n)$ additional memory. However, its performance warrantied further investigation into further reduction of memory requirements. The preliminary experiments indicate that C2-K is the most efficient version and so it is the one that should be the used as the basis for parallelization. Let us remark that variant C (and any of its modifications) could be used as an extension of any repetitions algorithm that reports repetitions of the same period together.

References

1. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*, J. Discr. Algorithms 2 (2004), pp. 53–86
2. G. Chen, S.J. Puglisi, and W.F. Smyth: *Fast & practical algorithms for computing all the runs in a string*, Proc. 18th Annual Symposium on Combinatorial Pattern Matching (2007), pp. 307–315
3. M. Crochemore: *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett. 5 (5) 1981, pp. 297–315
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press 2007
5. M. Crochemore and L. Ilie: *Maximal repetitions in strings*, Journal of Computer and System Sciences 74-5 (2008), pp. 796–807
6. M. Farach: *Optimal suffix tree construction with large alphabets*, 38th IEEE Symp. Found. Computer Science (1997), pp. 137–143
7. F. Franek, W.F. Smyth, and X. Xiao: *A note on Crochemore's repetitions algorithm, a fast space-efficient approach*, Nordic J. Computing 10-1 (2003), pp. 21–28
8. C. Iliopoulos, D. Moore, and W.F. Smyth: *A characterization of the squares in a Fibonacci string*, Theoret. Comput. Sci.,172 (1997), pp. 281-291
9. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, J. of Discrete Algorithms, (1) 2000, pp. 159–186
10. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, Proc. 30th Internat. Colloq. Automata, Languages & Programming (2003), pp. 943–955
11. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, Proc. 14th Annual Symp. Combinatorial Pattern Matching, R. Baeza-Yates, E. Chàvez, and M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003), pp. 200–210
12. M.G. Main: Detecting leftmost maximal periodicities, Discrete Applied Math., (25) 1989, pp. 145–153
13. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput. 22 (1993), pp. 935–938
14. P. Weiner: *Linear pattern matching algorithms*, Proc. 14th Annual IEEE Symp. Switching & Automata Theory (1973), pp. 1–11