An Improved Version of the Runs Algorithm Based on Crochemore's Partitioning Algorithm

Frantisek Franek*, Mei Jiang**, and Chia-Chun Weng

Department of Computing & Software
Faculty of Engineering
McMaster University
Hamilton, Ontario
Canada L8S 4K1
{franek,jiangm5,wengc2}@mcmaster.ca

Abstract. Though there are in theory linear-time algorithms for computing runs in strings, recently two of the authors implemented an $O(n \log n)$ algorithm to compute runs that was based on the Crochemore's partitioning repetitions algorithm. The algorithm preserved the running complexity of the underlying Crochemore's algorithm; however, the static memory requirement – already large at $14n$ integers for a string of length n – was increased significantly to $O(n \log n)$ integers. The purpose and advantage of this algorithm was its speed. In this paper we present a more advanced version of the extension of the Crochemore's algorithm for computing runs. This version in addition to maximal repetitions, computes runs and primitively rooted distinct squares. Its implementation completely does away with the extra memory required for the previous version and through some additional memory saving techniques, the overall memory need was reduced to $13n$ integers.

Keywords: repetition, run, distinct squares, string, periodicity, suffix array, LCP array, Lempel-Ziv factorization

1 Introduction

Crochemore's repetitions algorithm, often also referred to as Crochemore's partitioning algorithm, was introduced in 1981 [2] and was the first $O(n \log n)$ – and hence optimal – algorithm to compute maximal repetitions in a string of length n. The big advantage of the algorithm was its independence on the size of the alphabet of the string. Its disadvantage was in the implementation, as the data structures required for keeping track of the refinement process and the gaps require a substantial storage – originally estimated in about $20n$ of integers – and a complex machinery to update and maintain them. In 2003, Franek, Smyth, and Xiao [6] implemented the algorithm using several memory saving techniques lowering the requirement to $14n$ integers. An additional advantage of their implementation was that the memory was static, or to be more precise, allocated all at once at the outset of the algorithm as the working of the algorithm did not require any dynamic allocation or deallocation of memory. This approach led to an implementation with quite fast running times.

Since the advent of linear-time algorithms to compute suffix arrays [8,10,11], an avenue opened for true linear-time algorithms to compute runs.

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

** Supported in part by Queen Elizabeth II Graduate Scholarship in Science and Technology, and Ontario Graduate Scholarship Program

Frantisek Franek, Mei Jiang, Chia-Chun Weng: An Improved Version of the Runs Algorithm Based on Crochemore's Partitioning Algorithm, pp. 98–105.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2

Such algorithms follow the general strategy of

(a) compute suffix array using any of the linear-time algorithms, for instance [8,10,17,11,18],
(b) compute LCP (longest common prefix) array using any of the linear-time algorithms, for instance [9,16],
(c) compute Lempel-Ziv factorization using any of the linear-time algorithms, for instance [1,3],
(d) compute some runs that include all leftmost runs from the Lempel-Ziv factorization using Main's algorithm [14,15],
(e) from the runs computed in (d), compute all runs using Kolpakov-Kucherov's approach [12,13].

The laborious and circuitous strategy for linear-time algorithms suggests that performance of such algorithms may not be satisfactory. Franek and Jiang [4,5] extended the original Crochemore's repetitions algorithm to compute runs with a plan to benchmark the algorithm and compare it with any implementation of the linear-time algorithm for computing runs. Their implementation was based on Franek, Smyth, Xiao's implementation [6] for its optimized memory handling. The approach was quite straightforward: the maximal repetitions as reported were collected and consolidated into runs. This necessitated additional data structures of size $O(n \log n)$ integers. The program still exhibited fast running times, but the memory requirement was too substantial and required dynamic handling of memory during processing, which is quite a detriment to fast performance.

The reason to revisit the algorithm and modify it was to lower the memory requirement, eliminate the need of dynamic memory allocation and deallocation during processing, and prepare the stage for the parallelization. This report describes the new implementation that requires only a single allocation of $13n$ integers at the outset of the algorithm, preserves all the advantages of the previous implementations, computes not only the maximal repetitions – *as the original Crochemore's algorithm does*, but also the runs – *as the Franek and Jiang's implementation does*, and in addition it computes the number of primitively rooted distinct squares. Moreover, the algorithm in this form is well-posed for parallelization in the shared-memory model. We refer to this algorithm as FJW.

2 Preliminaries

For $e \geq 2$ and a non-empty string w, $(ww)^e$ is a *repetition of power* e in a string x if there are strings u and v, possibly empty, so that $x = u(ww)^e v$. w is referred to as the *generator* of the repetition, while the size of the generator is referred to as the *period* of the repetition. If $e = 2$, we talk of a *square*. A string is *primitive* if it is not a repetition. A repetition is *primitively rooted* if its generator is primitive. A repetition $(ww)^e$ in $x = u(ww)^e v$ is *maximal* if w is neither a suffix of u nor a prefix of v. For a string $x = x[0..n1]$, a repetition can be encoded as a triple (s, p, e), where s is the starting position of the repetition, p is the period, and e is the power.

A more succinct notion is that of a run. In a string $x = x[0..n1]$ a quadruple (s, p, e, t) encodes a *run* if

(a) for any $0 \leq i \leq t$, $(s{+}i, p, e)$ is a maximal repetition,
(b) either $s = 0$ or $(s1, p, 2)$ is not a square, i.e. the run cannot be extended to the left,

(c) either $s{+}t = n1$ or $(s{+}t{+}1, p, 2)$ is not a square, i.e. the run cannot be extended to the right,
(d) the generator $x[s..s{+}p1]$ is primitive.

The maximum number of maximal repetitions in a string of length n is $O(n \log n)$, see [2]. On the other hand, the maximum number of runs is $\leq 1.029n$, see [19]. In [4,5], Franek and Jiang used Crochemore's repetitions algorithm to generate all maximal primitively rooted repetitions, collect them in a data structure of size $O(n \log n)$ and then in $O(n \log n)$ time process the collected repetitions and consolidate them into runs. Though the repetitions computed by Crochemore's algorithm are not in any particular order – except the fact that repetitions of the same period are computed at the same stage, a detailed examination of the gap function revealed that there is no need to collect the repetitions, that the runs can be directly inferred from the information provided by the gap function.

To be able to discuss the gap function and show how the runs can be directly inferred, we need to briefly discuss the mechanism of the Crochemore's repetitions algorithm.

3 Brief description of Crochemore's repetitions algorithm

In mathematical terms, the algorithm is simple and elegant and relies on the refinements of classes of equivalence of the positions of the input string $x = x[0..n1]$. An equivalence $\approx_k$ is defined on the set of indices $\{0, \ldots, n1\}$ by $i_1 \approx_k i_2$ if and only if $x[i_1..i_1{+}k1] = x[i_2..i_2{+}k1]$. In simple terms, two positions are $\approx_k$ equivalent, if the substrings of length k starting at those two positions are the same. In all times, the algorithm maintains an ascending order of the indices in each class, though no particular order of the classes themselves.

At the first level, the algorithm computes by brute force the classes of equivalence $\approx_1$. These classes in fact represent all the positions with the same alphabet symbol. On each following level k, all classes of equivalence $\approx_k$ are computed. Note that each class from level $k1$ is either preserved as a class on level k, or is partitioned into several disjoint classes which we will refer to as *family*. That is why the Crochemore's algorithm is also referred to as the partitioning algorithm. It is clear that once a class has size 1, it cannot be partitioned any further. The processing ends when all classes are of size 1.

The classes, indeed, contain all information of all possible repeats of substrings of x. It is straightforward to see that a primitively rooted square of period p must be represented by two consecutive indices i_1 and i_2 in the same class of $\approx_p$ so that $|i_1 i_2| = p$.

The main complication of the algorithm lies in the process of refinements. If the refinements were carried out directly through references to the input string, the running complexity would be unacceptable $O(n^2)$. However, the refinement of the class on level k can be carried out by using other classes on level k which allows to discard the original string once the classes on the first level had been computed. This approach, though much better than the refinement through direct reference to the input string, would still lead to the running complexity of $O(n^2)$. If in each family we take a largest class by size and designate it *big* and all other as *small*, we can carry the full refinement of all the classes using just the small classes. Since any position can

occur in at most $O(\log n)$ small classes, this approach gives the running complexity of $O(n \log n)$.

Not to destroy the $O(n \log n)$ complexity, we cannot afford to scan the classes when looking for squares and ultimately for maximal repetitions. Throughout the whole process of refinement, a function $Gap(p)$ is maintained that gives a list of all indices that are exactly p distance from its predecessor in the class, more precisely: when processing level k, if $i_2 \in Gap(p)$, then $i_1 = i_2 p$ is in the same class of equivalence $\approx_k$ as i_2 and these two indices are consecutive in the class. We will describe the $Gap()$ function in more detail in the next section dealing with the implementation of the FJW algorithm.

4 Implementation of the FJW

We first describe the implementation and its data structures without any regard for the size of required memory. This leads to an implementation requiring $19n$ of integers. Then we use several techniques to reduce the required memory to $13n$ integers. We will present the data structures as static, but for practical reasons – we do not want to recompile the program each time a different string is to be processed, all the structures are allocated once at the outset of the program's processing. The structures are essentially arrays used to emulate doubly-linked lists, stacks, and queues.

The first seven arrays deal with classes:

1. An integer array $CStart\,[0..n1]$ stores the very first element of a class, i.e. $CStart[i] = j$ means that the first element of class i is j. *This emulates a pointer to the beginning of a class.*
2. An integer array $CEnd[0..n1]$ stores the very last element of a class, i.e. $CEnd[i] = j$ means that the last element of class i is j. *This emulates the pointer to the end of a class.*
3. An integer array $CNext[0..n1]$ stores the next element in the class or `null`. Thus $CNext[i] = j$ indicates that i and j are in the same class and that j is the next element after i, while $CNext[i] =$`null` indicates the i is the last element in the class. *This emulates the forward links.*
4. An integer array $CPrev[0..n1]$ stores the previous element in the class or `null`. Thus $CPrev[i] = j$ indicates that i and j are in the same class and that j is the element just before i, while $CPrev[i] =$`null` indicates the i is the first element in the class.*This emulates the backward links.*
5. An integer array $CMember[0..n1]$ stores the membership of each element, i.e. $CMember[i] = j$ means that i belongs to the class j.
6. An integer array $CSize[0..n1]$ stores the sizes of classes, i.e. $CSize[i] = j$ means that class i has size j.
7. An integer array $CEmpty[0..n1]$ is used as a stack of empty classes to be used.

The following four arrays deal with families:

1. An integer array $FStart[0..n1]$ is used as a stack. $FStart[i] = j$ thus means that class j is the first class in the family i.
2. An integer array $FNext[0..n1]$ emulates the forward links in a list of classes in a family.
3. An integer array $FPrev[0..n1]$ emulates the backward links.

4. An integer array $FMember\,[0\,..\,n1]$ stores the family membership, i.e. $FMember[i] = j$ means that class i belongs to family j.

The following four arrays deal with the refinement process:

1. An integer array $Refine[0..n1]$. $Refine[i] = j$ means that an element from class i should be moved to class j.
2. An integer array $RStack[0..n1]$ is used as a stack. It is used to remember which items in *Refine*[] were occupied, so it can be cleared without any need to traverse the whole array *Refine*[] which would destroy the $O(n \log n)$ complexity.
3. An integer array $Sel[]$ is used as a queue. It is the queue of all elements of all small classes.
4. An integer array $Sc[]$ is used as a queue of small classes. $Sc[i] = j$ indicates j is the last element of a small class. Thus the information in $Sel[]$ and $Sc[]$ implements a list of elements of small classes with indicators where one small class ends and the next small class starts.

The last four arrays implement the gap function:

1. An integer array $Gap[0..n1]$. $Gap[i] = j$ indicates that the first element in the gap list for i is j, i.e. j's predecessor in the class is ji.
2. An integer array $GMember[0..n1]$. $GMember[i] = j$ means that i belongs to the gap list j.
3. An integer array $GNext[0..n1]$ emulates the forward links in the gap lists.
4. An integer array $GPrev[0..n1]$ emulates the backward links in the gap lists.

The C++ code for this version is the file `crochB.cpp` and electronically available at [20].

In the next version, `crochB1.cpp`, also posted at [20], the array $GMember[]$ is replaced by a function $GMember()$ and the memory requirement is reduced to $18n$ integers. $GMember()$ can be directly computed:

$$GMember(i) = \begin{cases} \texttt{null} & \text{if } i \text{ is not member of any class,} \\ \texttt{null} & \text{if } i \text{ is the first member of a class,} \\ iCPrev[i] & \text{otherwise.} \end{cases}$$

Version `crochB3.cpp` reduces the memory requirement further to $17n$ integers. Consider any family doubly-linked list, its beginning can be determined by two means: $FStart[i] = j$ or $FPrev[j] =$`null`. Thus, we can do away with $FMember[]$ array and replace it by a function that is utilizing the redundant space in $FStart[]$ and $FNext[]$:

$$FMember(i) = \begin{cases} FStart[i] & \text{if the stack pointer is null,} \\ FNext[FPrev[FStart[i]]] & \text{if } i \leq \text{the stack pointer,} \\ FStart[i] & \text{otherwise.} \end{cases}$$

We also introduce a function $FEnd()$ computed from $FStart[]$ and $FPrev[]$: $FEnd(i) = FPrev[FStart[i]]$.

In the next version, `crochB4.cpp`, $CEmpty[]$ and $Sc[]$ are made to share the same memory segment, reducing the memory requirement to $16n$ integers.

Version `crochB5.cpp` distributes $CEnd[]$ and $CSize[]$ over $CStart$, $CNext$, and $CPrev$, thus reducing the memory requirement further to $14n$ integers. Therefore, $CEnd(i) = CPrev[CStart[i]]$ and $CSize(i) = CNext[CPrev[CStart[i]]]$.

If we limit the maximal possible length of an input string from `UNSIGNED LONG MAX` to `LONG MAX`, which for a 32-bit long it is 2,147,483,647 and thus large enough, we can virtualize $CMember[]$ over $Gap[]$, $GNext[]$, and $GPrev[]$, reducing the memory requirement to $13n$ integers. Thus, the function to set the value of $CMember(e)$ to c:

```
if (Gap[e] == null || Gap[e] < 0)
     if (c == null)
         Gap[e] = null;
     else
         Gap[e] = 0-1-c;
else
    if (c == null)
        GNext[GPrev[Gap[e]]] = null;
    else
        GNext[GPrev[Gap[e]]] = 0-1-c;
```

and the function to get the value of $CMember(e)$:

```
if (Gap[e]==null)
     return null;
else
     if (Gap[e] < 0)
         return 0-1-Gap[e];
     else
         if (GNext[GPrev[Gap[e]]] == null)
             return null;
         else
             return 0-1-GNext[GPrev[Gap[e]]];
```

The version `crochB7.cpp` is just a polished version of `crochB6.cpp` with the additional features discussed in the next section.

5 The gap function and computations of distinct squares, maximal repetitions, and runs

Throughout the process of refinement, the gap function is maintained. In order to protect the running complexity of $O(n \log n)$, every time an element is removed from a class, the gap function is updated; and any time an element is added to a class, the gap function is updated again. When computing the next level from the current one, $Gap[p]$ points to the first element whose immediate predecessor in its class is exactly at distance p, while $GNext[]$ and $GPrev[]$ allow us to traverse the whole list in either direction and to update the list in constant time. Notice that if $Gap[p] = i$ and we are dealing with level p of refinement, then there is a primitively rooted square starting at position $GPrev[i]$ of period p.

Computing Distinct Squares – `traceSquares()` method

The gap function can be used to compute primitively rooted distinct squares. As we traverse the gap list, once we identify the first primitively rooted square in each class,

we ignore the identification of the rest from the same class as they are all identical squares. We use *Refine*[] and *RStack*[] that are only needed during the refinement process as auxiliary data structures here to indicate the last class we already have a representative from in order not to get another representative from the same class. Note that the program can either output the number of distinct squares, the triples (s, p, e) encoding the squares identified, or the squares as identified substrings of the input string – we refer to is as *pretty print*. To use *pretty print*, the string alphabet should be the lower case letters $a, b, \ldots$

Computing Maximal Repetitions – `traceMaxReps()` method

For the maximal primitively rooted repetitions, again either their number can be output, the individual repetitions in their encoding into triples or *pretty print* can be used. The algorithm traverses the gap list, and for each entry it checks how far left and how far right it can extend the square. Thus, during the tracing at level p, all the individual squares identified are consolidated into maximal repetitions. A brief description on how the algorithm determines if the square can be extended to the left: the entry i from the gap list $Gap[p]$ indicates that there is a primitively rooted square starting at position ip. Then the algorithm checks if the square can be extended to the left – i.e. is there a square of period p starting at position $i2p$ and determined by ip. It is possible that the position ip is in the gap list further away. In order not to process the square starting at $i2p$ and determined by ip, we again use *Refine*[] and *Rstack*[] to indicate that this entry has already been processed.

Computing Runs – `traceRuns()` method

The computation of runs is performed by $TraceRuns()$. The idea is very similar to that of tracing maximal repetitions: the identified primitively rooted squares are consolidated to runs. If you look at the leading square of a run (s, p, e, t) that must be primitively rooted by definition, at every position $s{+}i$, $0 \leq i \leq (e2) \cdot p{+}t$ there is a primitively rooted square. This fact is based on a simple observation that a rotation of a primitive string is also primitive. In the algorithm, we have to consolidate the run from all of the primitively rooted squares encoded in the gap function. Thus, having identified a square, not only we must check if it can be extended left or right as a repetition, we have to check if it can be shifted left or right. Again, we are using *Refine*[] and *RStack*[] as auxiliary data structures to indicate which of the elements of the gap list had been previously processed as the part of tracing, so we do not process them again.

6 Conclusion

We present a new implementation of an extension of the Crochemore's repetitions algorithm that computes primitively rooted distinct squares, primitively rooted maximal repetitions, or runs. The running complexity of the original repetitions algorithm is preserved, and thus is $O(n \log n)$ where n is the length of the input string. In comparison to the previous implementation of the Crochemore's partitioning algorithm, the memory required is reduced to $13n$ integers. In comparison to the previous implementation of an extension to compute runs, there is no additional memory requited and no dynamic allocation or deallocation of the memory during the processing as

all the required memory is allocated once at the outset of the program. The resulting algorithm implemented in C++ is very fast and all the versions described in this paper can be downloaded from [20]. Since this report does not include benchmarking and comparisons with other runs algorithms, the future work must include the bench-marking and comparisons with the fastest algorithms [1,7] regardless their complexity, and, of course, with the known linear implementations.

References

1. G. Chen, S. J. Puglisi, and W. F. Smyth: *Fast and practical algorithms for computing all the runs in a string*, in Proc. the 18th annual symposium on Combinatorial Pattern Matching (CPM 2007), 2007, pp. 316–327.
2. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inform. Process. Lett., 12(5) 1981, pp. 297–315.
3. M. Crochemore, L. Ilie, and W. F. Smyth: *A simple algorithm for computing the lempel-ziv factorization*, in Proc. the 17th Data Compression Conference (DCC 2008), 2008, pp. 482–488.
4. F. Franek and M. Jiang: *Crochemore's repetitions algorithm revisited – computing runs.* to appear in Int. Jour. of Foundations of Comp. Sci.
5. F. Franek and M. Jiang: *Crochemore's repetitions algorithm revisited – computing runs*, AdvOL Report 2009/11, Advanced Optimization Laboratory, Dept. of Comp. and Software, McMaster University, 2009.
6. 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.
7. K. Hirashima, H. Bannai, W. Matsubara, A. Ishino, and A. Shinohara: *Bit-parallel algorithms for computing all the runs in a string*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 203–213.
8. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. 30th Internat. Colloq. on Automata, Languages & Programming (ICALP 2003), no. 2719 in LNCS, Springer, 2003, pp. 943–955.
9. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Symposium on Combinatorial Pattern Matching (CPM 2001), no. 2089 in LNCS, Springer, 2001, pp. 181–192.
10. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proc. the 14th Annual Conference on Combinatorial Pattern Matching (2003), no. 2676 in LNCS, Springer, 2003, pp. 186–199.
11. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. 14th Annual Symp. Combinatorial Pattern Matching, R. Baeza-Yates, E. Chàvez, and M. Crochemore, eds., no. 2676 in LNCS, Springer, 2003, pp. 200–210.
12. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1999, pp. 596–604.
13. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in Proc. 12th Intl. Symp. on Fund. of Comp. Sci. 1999, no. 1684 in LNCS, 1999, pp. 374–385.
14. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Maths. – Combinatorics and Complexity, 25(1–2) 1989, pp. 145–153.
15. M. G. Main and R. J. Lorentz: *An $O(n \log n)$ algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
16. G. Manzini: *Two space saving tricks for linear time LCP array computation*, in Proc. SWAT 2004, no. 3111 in LNCS, Springer, 2004, pp. 372–383.
17. G. Nong, S. Zhang, and W. H. Chan: *Linear time suffix array construction using d-critical substrings*, in Proc. 20th Combinatorial Pattern Matching (CPM 2009), Lille, France, June 2009, pp. 54–67.
18. S. Zhang and G. Nong: *Fast and space efficient linear suffix array construction*, in Proc. IEEE Data Compression Conference (IEEE DCC), IEEE Computer Society, March 2008, p. 553.
19. `http://www.csd.uwo.ca/faculty/ilie/runs.html`.
20. `http://www.cas.mcmaster.ca/~franek/research.html`.