A note on Crochemore's repetitions algorithm a fast space-efficient approach[1]

Frantisek Franek[1], W. F. Smyth[1,2], and Xiangdong Xiao[1]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
(www.cas.mcmaster.ca/cas/research/groups.html)
[2] School of Computing, Curtin University, GPO Box U-1987
Perth WA 6845, Australia

e-mail: franek@mcmaster.ca, smyth@mcmaster.ca

August 12, 2002

Abstract. The space requirement of Crochemore's repetitions algorithm is generally estimated to be about $20MN$ bytes of memory, where N is the length of the input string and M the number of bytes required to store the integer N. The same algorithm can also be used in other contexts, for instance to compute the suffix tree of the input string in $O(N \log N)$ time for the purpose of data compression. In such contexts the large space requirement of the algorithm is a significant drawback. There are of course several newer space-efficient algorithms with the same time complexity that can compute suffix trees or arrays. However, in actual implementations, these algorithms may not be faster than Crochemore's. Therefore, we consider it interesting enough to describe a new approach based on the same mathematical principles and observations that were put forth in Crochemore's original paper, but whose space requirement is $10MN$ bytes. Additional advantages of the approach are the ease with which it can be implemented in C/C++ (as we have done) and the apparent speed of such an implementation in comparison to other implementations of the original algorithm.

1 INTRODUCTION

Crochemore's algorithm [C81] computes all the repetitions in a finite string $\boldsymbol{x}$ of length N in $O(N \log N)$ time. The algorithm in fact computes rather more and can be used, for instance, to compute the suffix tree of $\boldsymbol{x}$, hence possibly as a tool for expressing $\boldsymbol{x}$ in a compressed form. In such contexts the space requirement becomes as important as the time complexity. It appears that known implementations of Crochemore's algorithm require at least $20MN$ bytes of memory for the task of refining the equivalence classes alone, where M is the number of bytes required to store the integer N.

Here we present a different implementation based on the mathematical properties and observations of [C81] and thus having the same time complexity $O(N \log N)$ as

[1]Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

the original algorithm. However, the new data structures used for the representation of classes and for the execution of the refinement process allow the space requirement to be substantially reduced.

There are several newer space-efficient algorithms to compute suffix trees or arrays (notably [U92], [MM93]) of the same worst-case complexity as Crochemore's. The motivation for our investigation of a space-efficient implementation of the classical Crochemore's algorithm that may be competetive with these newer algorithms stems from the fact that the actual implementations of these algorithms may not in fact be any faster.

A large memory saving comes from the fact that our algorithm requires storage for only N classes at any given time, rather than $2N$ as in the original algorithm. This alone brings the space requirement down to $15MN$. Of course there is some extra processing related to this reduction in space, but it does not affect the time complexity, and in fact it appears that in practice our implementation runs a good deal faster than the standard implementation proposed in [C81]. A further $5MN$ space reduction is achieved by smart utilization of the space:

- allowing space to be shared by data structures, as in memory multiplexing — for example, if a queue empties faster than a stack grows, then they can share the same memory segment;

- spreading one data structure across several others, as in memory virtualization.

Taken together, these "tricks" bring the space requirement down to $10MN$.

Additional advantages of this approach are the ease with which it can be implemented in C/C++ (as we have done) and, as remarked above, its apparent speed in comparison to other implementations of the original algorithm.

In this paper we do not due to space limitations provide any detailed computer instructions, but we try to give a high-level description of our approach, so that the reader can understand how the space savings are achieved.

In our discussion below we assume that the reader is familiar with both Crochemore's algorithm and its mathematical foundation. We make the usual assumption required for Crochemore's algorithm that the alphabet is ordered; therefore we are able to assume further that the classes corresponding to the first level ($p = 1$) can be computed in $O(N \log N)$ time.

For better comprehension, we present the algorithm in two stages. The first stage, FSX15 (with space requirement $15MN$ bytes), exhibits all important procedural and control aspects of our algorithm without the complications of memory multiplexing and virtualization. Then the second stage, FSX10, incorporates the changes required by memory multiplexing and virtualization to reduce the space requirement to $10MN$. Finally, we present some rough results of computer runs that compare the time and space requirements of our approach with those of a standard implementation of Crochemore's algorithm.

2 DATA STRUCTURES FOR FSX15

Recall that for each $p = 1, 2, \ldots,$ Crochemore's algorithm acts on a given string $\boldsymbol{x} = \boldsymbol{x}[1..N]$ to compute equivalence classes $\{i_1, i_2, \ldots, i_r\}$, where for every $1 \leq j < h \leq r$,

$$\boldsymbol{x}[i_j..i_j{+}p{-}1] = \boldsymbol{x}[i_h..i_h{+}p{-}1].$$

The positions i_j in each class are maintained in increasing sequence: $i_j < i_{j+1}$, $1 \leq j < r$. At each step of the algorithm, each class $\mathcal{C}_p$ that is not a singleton is decomposed into a ***family*** of subclasses $\mathcal{C}_{p+1,s}$; of these subclasses, the one of largest cardinality is called ***big***, the others are ***small***. A straightforward approach to this decomposition would require order N^2 time in the worst case, but Crochemore's algorithm reduces this time requirement by carrying out the decomposition from p to $p{+}1$ only with respect to the small classes identified at step p. Since each position can belong to a small class only $O(\log N)$ times, it follows that the total time requirement is $O(N \log N)$. As remarked in the introduction, we may assume that the classes corresponding to $p = 1$ have initially been computed in $O(N \log N)$ time. Note that the version of Crochemore's algorithm discussed here does not explicitly compute repetitions; we will be interested only in reducing each of the equivalence classes to a singleton.

We will use an integer array of size N to represent the classes computed at step p. We have several requirements:

- we need to keep the elements of the classes in ascending order;
- we need an efficient way to delete any element (so that we need to represent each class as a doubly-linked list);
- we need an efficient way to insert a new element at the end of a class (and hence we need a link to the last element of the class);
- we need efficient access to the size of a class;
- we need efficient access to a class (and hence we need a link to the first element of the class);
- last but not least, we need an efficient way to determine to which class a given element belongs.

To satisfy all these requirements, we use six integer arrays of size N:

- `CNext[1..N]` emulates forward links in the doubly-linked list. Thus $\texttt{CNext}[i] = j > i$ means that j is the next element (position) in the class that i belongs to. If there is no position $j > i$ in the class, then $\texttt{CNext}[i] = \texttt{null}$.
- `CPrev[1..N]` emulates backward links in the doubly-linked list. Thus $\texttt{CPrev}[i] = j < i$ means that j is the previous element (position) in the class that i belongs to. If there is no position $j < i$ in the class, then $\texttt{CPrev}[i] = \texttt{null}$.
- `CMember[1..N]` keeps track of membership. Thus $\texttt{CMember}[i] = k$ means that element i belongs to the class with index k ($i \in c_k$), while $\texttt{CMember}[i] = \texttt{null}$ means that at this moment i is not member of any class.

- `CStart[1..N]` keeps links to the starting (smallest) element in each class. Thus $\texttt{CStart}[k] = i$ means that the class c_k starts with the element i, while $\texttt{CStart}[k] = \texttt{null}$ means that at this moment the class c_k is empty.
- `CEnd[1..N]` keeps links to the final (largest) element in each class. Thus $\texttt{CEnd}[k] = i$ means that the class c_k ends with the element i; the value of $\texttt{CEnd}[k]$ is meaningful only when $\texttt{CStart}[k] \neq \texttt{null}$.
- `CSize[1..N]` records the size of each class. Thus $\texttt{CSize}[k] = r$ means that the class c_k contains r elements; the value of $\texttt{CSize}[k]$ is meaningful only when $\texttt{CStart}[k] \neq \texttt{null}$.

Suppose that there exists a class $c_3 = \{4, 5, 8, 12\}$, indicating that the substrings of length 3 beginning at positions $4, 5, 8, 12$ of $\boldsymbol{x}$ are all equal. Then c_3 would be represented as follows:

$$\texttt{CNext}[4] = 5,\ \texttt{CNext}[5] = 8,\ \texttt{CNext}[8] = 12, \texttt{CNext}[12] = \texttt{null};$$
$$\texttt{CPrev}[12] = 8,\ \texttt{CPrev}[8] = 5,\ \texttt{CPrev}[5] = 4,\ \texttt{CPrev}[4] = \texttt{null};$$
$$\texttt{CMember}[4] = \texttt{CMember}[5] = \texttt{CMember}[8] = \texttt{CMember}[12] = 3;$$
$$\texttt{CStart}[3] = 4;\ \ \texttt{CEnd}[3] = 12;\ \ \texttt{CSize}[3] = 4.$$

We need to track the empty classes, and for that we need a simple integer stack of size N, `CEmptyStack`, that holds the indexes of the empty (and hence available) classes. This stack, as well as all other list structures used by Crochemore's algorithm, is implemented as an array that requires MN bytes of storage. Such an approach saves time by allowing all space allocation to take place only once, as part of program initialization. We introduce two operations on the stack, ***CEmptyStackPop()*** that removes the top element from the stack and returns it, and ***CEmptyStackPush(i)*** that inserts the element i at the top of the stack.

We shall process classes from one refinement level p to the next level $p+1$ by moving the elements from one class to another, one element at a time. We view the classes as permanent containers and distribute the elements among them, so that at any given moment we need at most N classes. This means that the configuration of classes at level p is destroyed the moment we move a single element. But, as we shall see, we do not really need to keep the old level intact if we preserve an essential "snapshot" of it before we start tinkering with it.

What we need to know about level p will be preserved in two queues, `SElQueue` and `SCQueue`. `SElQueue` contains all the elements in small classes in level p, organized so that the elements from the same small class are grouped together in the queue and stored in ascending order. `SCQueue` contains the first element from each small class, thus enabling us to identify in `SElQueue` the start of each new class. Therefore, when these queues are created, we must be careful to process the small classes of level p in the same order for both of them. For instance, if level p had three small classes,

$$c_3 = \{2, 4, 5, 8\},\ c_0 = \{3, 6, 7, 11\},\ c_5 = \{12, 15\},$$

`SElQueue` could contain $2, 4, 5, 8, 3, 6, 7, 11, 12, 15$ in that order, while the corresponding `SCQueue` would contain $2, 3, 12$. The order of the classes (c_3 followed by c_0 followed by c_5) is not important; what is important that the same order is used in order to create `SElQueue` and `SCQueue`. After the two queues have been created, we do not need level p any more and we can start modifying it. Of course we suppose that we have available the usual queue operations:

- ***SElQueueHead()*** (remove the first element from the queue and return it);
- ***SElQueueInsert(i)*** (insert the element i at the end of the queue);
- ***SElQueueInit()*** (initialize the queue to empty).

Analogous operations are available also for `SCQueue`.

When refining class c_k in level p using an element i from class $c_{k'}$, we might need to move element i–1 from c_k to a new or an existing class. To manage this processing, we keep an auxiliary array of size N, `Refine[1..N]`. Initially, when we start using the class $c_{k'}$ for refinement, all entries in `Refine[ ]` are `null`. If a new class c_h is created in level $p+1$ by moving $i-1$ out of class c_k and into c_h as its first element, we set `Refine`$[k] \leftarrow h$. If later on we move another element from c_k as a result of refinement by the same class $c_{k'}$, we use the value `Refine`$[k]$ to tell us where to move it to. This requires that when we start refining by a new class, we have to restore `Refine[ ]` to its original `null` state. Since we cannot afford to traverse the whole array `Refine[ ]` without destroying the $O(N \log N)$ time complexity, we need to store a record of which positions in `Refine[ ]` were previously given a non-`null` value. For this we make use of a simple stack, `RefStack`: every assignment to `Refine`$[k]$ causes the index k to be pushed onto the stack `RefStack`. As before, we assume that we have available the usual stack operations ***RefStackPop()*** and ***RefStackPush(i)***.

Since after completing the refinement of the classes in level p, we must determine the small classes in level $p+1$, we therefore need to maintain throughout the refinement process certain families of classes (to be more precise, families of class indexes). As noted above, a family consists of the classes in level $p+1$ that were formed by refinement of the same class in level p. A family may or may not include the original class from level p itself (it may completely disappear if we remove all its elements during the refinement). We need an efficient way to insert a new class in a family (the order is not important), an efficient way to delete a class from a family, and finally an efficient way to determine to what family (if any) a class belongs. These facilities can be made available by representing the families as doubly-linked lists implemented using arrays, just as we did previously with the classes themselves. In this case, however, the `Size[ ]` and `End[ ]` arrays are not required, so we can get by with only four arrays, as follows:

- `FNext[1..N]` emulates the forward links (as in `CNext[ ]`).
- `FPrev[1..N]` emulates the backward links (as in `CPrev[ ]`).
- `FMember[1..N]` keeps track of membership (as in `CMember[ ]`). Whenever `FMember`$[i] =$ `null`, it means that c_i is not a member of any family.
- `FStart[1..N]` gives the first class in each family (as in `CStart[ ]`).

Note that classes in families do not need to be maintained in numerical order, as was true earlier of positions in classes.

To summarize, in order to implement Crochemore's algorithm, it is sufficient to allocate 15 arrays, each of which provides storage space for exactly N integers of length M, thus altogether $15MN$ bytes of storage: `CNext`, `CPrev`, `CMember`, `CStart`, `CEnd`, `CSize`, `CEmptyStack`, `SElQueue`, `SCQueue`, `RefStack`, `Refine`, `FStart`, `FNext`, `FPrev`, and `FMember`.

3 DATA STRUCTURES FOR FSX10

As the first step in reducing the space complexity further, we are going to eliminate the `CSize[ ]` and `CEnd[ ]` arrays. For the very first element, say s, in a class, `CPrev[`s`]`= `null`, while for the very last element, say e, `CNext[`e`]`= `null`. But we have another way to discern the beginning of the class (`CStart[ ]`), so that `CPrev[ ]` becomes superfluous. Thus we can store `CPrev[`s`]`$\leftarrow e$, a direct link to the end of the class. This yields an efficient means to discern the end of the class, and so we can store in `CNext[`e`]` the size of the class. Hence `CPrev[CStart[`j`]]` takes on the role of `CEnd[`j`]`, while `CNext[CPrev[CStart[`j`]]]` takes on the role of `CSize[`j`]`. This is straightforward and the code need only be slightly modified to accommodate it. All we have to do is make sure that when inserting or deleting an element in or from a class, we update properly the end link and the size. When traversing a class, we have to make sure that we properly recognize the end (we cannot rely on the `null` value to stop us as in `FSX15`). We have in fact "virtualized" the memory for `CEnd[ ]` and `CSize[ ]`, and so reduced the space complexity to $13MN$.

When we take an element from `SElQueue` and use it for the purpose of refinement, at most one new class is created and thus at most one location of `Refine[ ]` is updated. This simple observation allows `RefStack` and `SElQueue` to share the same memory segment, as long as we make sure that `RefStack` grows from left to right, while the queue is always right justified in the memory segment. The changes in the code required to accommodate this are not very great — all we have to do is to determine before filling `SElQueue` what position we have to start with. In essence, we have "multiplexed" the same memory segment and brought the space complexity down to $12MN$.

The number of elements in `SCQueue` is the same as the number of small classes, which is less than or equal to the number of non-empty classes; thus the size of `SCQueue` plus the size of `CEmptyStack` at any given moment is at most N. This simple observation allows `CEmptyStack` and `SCQueue` to share the same memory segment, as long as we make sure that `CEmptyStack` is growing from left to right, while the queue is always right justified in the memory segment. Again, as above, the changes in the code required to accommodate this are not major. We again have "multiplexed" the same memory segment and brought the space complexity down to $11MN$.

The final memory saving comes from the fact that `FPrev[ ]` for the very first class in a family and `FNext[ ]` for the very last class in the same family are set to `null` and hence redundant for the same reasons as described above for `CPrev[ ]` and `CNext[ ]`. We can thus "virtualize" the memory for the array `Refine[ ]`. We will have to index it in reverse and we will use all the unused slots in `FStart[ ]` (i.e. slots with indexes > `FStartTop`) as well as the unnecessary `FNext[ ]` slots. The formula is rather simple. Instead of storing r in `Refine`$[i]$, we will use

```
SetRefine(i,r)
j←N-(i+1)
if FStartTop = null OR j > FStartTop then
  FStart[j]←r
else
  FNext[FPrev[FStart[j]]]←r
end SetRefine
```

and instead of fetching a value from `Refine[i]` we will use

```
integer GetRefine(i)
j←N-(i+1)
if FStartTop = null OR j > FStartTop then
  return FStart[j]
else
  return FNext[FPrev[FStart[j]]]
end GetRefine
```

The modification of the code is more complex in this case, since we have to track the ends of the family lists as we do for class lists; more importantly, when a new family is created, we have to save the `Refine[ ]` value stored in that so-far-unused slot k that now is going to be occupied by the start link of the family list, and store k at the end of the list instead. This "virtualization" of the memory for `Refine[ ]` brings the space complexity down to the final value of $10MN$.

4 INFORMATIVE EXPERIMENTAL RESULTS

To estimate the effect of our space reduction on time requirement, we have implemented two versions of Crochemore's algorithm:

- a naïve array-based version, FSX20, that executes Crochemore's algorithm using 20 arrays each of length N words:
- a version of FSX10 that requires 10 arrays each of length N words.

Thus both of these implementations are word-based: assuming a word-length of 32 bits, the value of M is actually fixed at 4.

We expect that FSX20 will execute Crochemore's algorithm about as fast as it can be executed, but at the cost of requiring exactly $20N$ words of storage. A version that implemented standard list-processing techniques rather than arrays to handle the queues, stacks and lists required by Crochemore's algorithm would generally require less storage: $11N$ words for arrays plus a variable amount up to $13N$ for the list structures. However, as a result of the time required for dynamic space allocation, such a version would certainly run several times slower than FSX20.

We must remark at this point that the experiments performed have only an informative value, for we conducted them without controlling many aspects depending on the platform (as memory caching, virtual memory system paging etc.), nor did we perform a proper statistical evaluation to control for other factors not depending on the platform (load on the machine, implementation biases etc.) Thus, we really do not claim any significant conclusions for the actual algorithms whose implementations were tested.

We have run FSX20 and FSX10 against a variety of long strings (up to 3.8 million bytes): long Fibonacci strings, files from the Calgary corpus, and others. The results indicate that FSX10 seems to require 20-30% more time than FSX20, in most cases a small price to pay for a 52% reduction in space.

References

[C01] Calgary Corpus

`http://links.uwaterloo.ca/calgary.corpus.html`

[C81] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *IPL 12-5* (1981) 244-250.

[MM93] Udi Manber & Gene W. Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM J. Comput. 22-5* (1993) 935-948.

[U92] Esko Ukkonen, **Constructing suffix trees on-line in linear time**, *Proc. IFIP 92*, vol. I (1992) 484-492.