Crochemore’s algorithm for repetitions revisited - computing runs

F. Franek, M. Jiang
Computing and Software
McMaster University
Hamilton, Ontario

Israel Stringology Conference
Bar-Ilan University, Tel-Aviv
March-April 2009
• Why we are interested in Crochemore’s repetition algorithm

• A brief description of our implementation of Crochemore’s algorithm.

• A simple modification of Crochemore’s algorithm to compute runs (worsening the complexity to $O(n \log^2(n))$)

• A modification of Crochemore’s algorithm to compute runs while preserving the complexity $O(n \log(n))$

• Conclusion
Why we are interested in Crochemore’s repetition algorithm
A **run** captures the notion of a maximal non-extendible repetition in a string $x$

$$(s,p,e,t)$$

Alternative: $(s,p,end)$

$$\begin{align*}
e &= (\text{end} - s + 1) / p \\
t &= (\text{end} - s + 1) \mod p
\end{align*}$$

- $s$: **starting position** (leftmost)
- $p$: **period**
- $e$: **power, exponent**
- $t$: **tail** (rightmost)
- **irreducible generator**
Computing runs in linear time

Main (1989) introduced runs and gave the following algorithm to compute the leftmost occurrence of every run of a string $x$:

(1) Compute a suffix tree for $x$ (*linear, using Farach’s algorithm*)

(2) using the suffix tree, compute Lempel-Ziv factorization of $x$ (*linear, Lempel-Ziv*)

(3) using the Lempel-Ziv factorization, compute the leftmost runs (*linear, Main*)
Lempel-Ziv factorization can be computed in linear time using suffix array (Abouelhoda, Kurtz, & Ohlebusch 2004)

Suffix array can be computed in linear time (Kärkkäinen, Sanders 2003, Ko, Aluru 2003)

All these approaches are complicated and elaborate, and the implementations into code are not readily available.

Also, they do not lend themselves well to parallelization (see slide 9 -- the refinement of the classes can be done naturally in parallel as the refinement of one class is independent from the refinement of another class.)

We have a good and “space efficient” implementation of Crochemore’s algorithm, that naturally lends itself to parallelization.
A brief description of our implementation of Crochemore’s algorithm
CNext[ ]  \( c_1 = \{2, 4, 5\} \)

CPrev[ ]

CEnd[ ]

CStart[ ]

CSize[ ]

Total this slide 6*N
subtotal 6*N

indexes

Israel Stringology Conference, Bar-Ilan
CEmptyStack
SelQueue
ScQueue
RefStack
Refine[]

indexes

0 1 3 .... N

0 1 3

CEmptyStack
SelQueue
ScQueue
RefStack
Refine[]

Total this slide 5*N
subtotal 11*N
indexes

FNExt[ ] \hspace{1cm} f_2 = \{3, 5\}

FPrev[ ]

FStart[ ]

FMember[]

Total this slide 4*N
overall total 15*N
$c_1 = \{2, 4, 5\}$

Memory virtualization

Total this slide $4*N$
subtotal $4*N$
CEmptyStack → ScQueue

RefStack → SelQueue

Refine[] is virtualized over FNext[], FPrev[], and FStart[]

Total this slide 2*N subtotal 6*N
Refine[] is virtualized over Memory virtualization.

Total this slide 4*N overall total 10*N
Total this slide 4*N
overall total 14*N
Though the repetitions are reported level by level, they are not reported in any appreciable order (caused by the manipulations of GapList)

```
ababababababab
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

- (10,1,2) 
- (7,1,2) 
- (2,1,2) 
- (11,2,2) 
- (3,2,2) 
- (4,2,2) 
- (6,3,2) 
- (5,3,3) 
- (0,3,2) 
- (7,3,2) 
- (0,5,2) 
- (1,5,2)

run run run
A simple modification of Crochemore’s algorithm to compute runs (worsening the complexity to $O(n \log^2(n))$)
We have to collect repetitions and “join” them into runs.
Collecting, “joining”, and reporting level by level, basically in a binary search tree:

- RunLeft[ ] (reuse FNext[ ])
- RunRight[ ] (reuse FPrev[ ])
- Run_s[ ] (reuse FMember[ ])
- Run_end[ ] (reuse FStart[ ])

Complexity: need $O(\log(n))$ for each repetition to place it in the tree, overall $O(n \log^2(n))$
Collecting and “joining” in a binary search tree, reporting at the end: the same complexity $O(n \log^2(n))$, memory requirement increased by $5*N$

Points to the “root” of the search tree for runs of period $p$. 

Israel Stringology Conference, Bar-Ilan
A modification of Crochemore’s algorithm to compute runs while preserving the complexity $O(n \log(n))$
Collecting into buckets, “joining” and reporting at the end.

Memory: \( O(n \log(n)) \)

Linked list of repetitions starting at \( s \)

\( p_2 \) points to the last run with period \( p_2 \), so we know with what to join the incoming repetition with (if at all), as we sweep from left to right.
Complexity: $O(n \log(n))$
Memory: $15N + O(n \log(n))$
To avoid dynamic allocation of memory, we are using allocation from arena technique.
Conclusion

• Crochemore’s algorithm is fast, though memory demanding

• Our implementation is as memory efficient as possible

• Great potential for parallel implementation

• Preliminary test very positive

• Further research
  (1) to compare performance with linear time algorithms (problem - lack of code)
  (2) to implement parallel version with little communication overhead
Thank You!