McMaster University Advanced Optimization Laboratory

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

An improved version of the runs algorithm based on Crochemore's partitioning algorithm

F. Franek, M. Jiang, and C. Weng

Advanced Optimization Laboratoty
Dept. of Computing and Software
McMaster University

Prague Stringology Conference
Aug. 29-31, 2011

Outline

1. Motivation
2. Background
3. The algorithm
4. Conclusion

Motivation

The purpose of this short note is twofold:

- to make researchers in the field aware that a `C++` implementation of a reasonably fast and reasonably efficient algorithm to compute runs, maximal repetitions, or distinct squares in a string is available (*can be downloaded*).
- solicit other `C/C++` implementations of runs algorithms for potential comparison and bench-marking.

Background

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

Since the advent of linear-time algorithms to compute suffix arrays

Kärkkäinen+Sanders 2003
Kim+Sim+Park+Park 2003
Ko+Aluru 2003

an avenue opened for true linear-time algorithms to compute runs. Such algorithms follow the same general strategy:

Background, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

(a) compute suffix array using any of the linear-time algorithms

(b) compute LCP (longest common prefix) array using any of the linear-time algorithms

(c) compute Lempel-Ziv factorization using any of the linear-time algorithms

(d) compute some runs that include all leftmost runs from the Lempel-Ziv factorization using Main's algorithm

(e) from the runs computed in (d), compute all runs using Kolpakov-Kucherov's approach

Background, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

This is laborious and circuitous, and as Smyth et. al. showed, does not lead to particularly fast or memory efficient algorithms.

Background, *cont.*

Franek+Jiang 2009 extended Crochemore's repetitions algorithm to compute runs. The extension was based on a memory efficient implementation of Crochemore's algorithm by Franek+Smyth+Xiao 2003 and required additional data structures of $O(n \log n)$ integers while preservening the original worst-time complexity of $O(n \log n)$.

Background, *cont.*

The algorithm was straighforward:

- collect the maximal repetitions as output by the underlying Crochemore's algorithm
- consolidate them into runs

Three variants produced differing the strategy in when and how the collected maximal repetitions are consolidated into runs.

Background, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

Our computational investigation of the *maximum-number-of-runs* conjecture and the *maximum-number-of-distinct-squares* conjecture using the *d*-step approach required a fast and memory efficient computer program. The new and improved algorithm was developed to satisfy this need.

The algorithm

Partitioning-based Runs Algorithm
Motivation
Background
Algorithm
Conclusion

We skip the description of the implementation details that use mupltiplexing and other various methods to reduce the memory requirement to $13n$ integers, where n is the size of the input string.

Moreover, all memory is allocated as a single segment prior to the processing and no other dynamic memory allocation/deallocation takes place.

The algorithm, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

Crochemore's partitioning algorithm computes level-by-level using refinement the classes of equivalence and maintains a gap function.

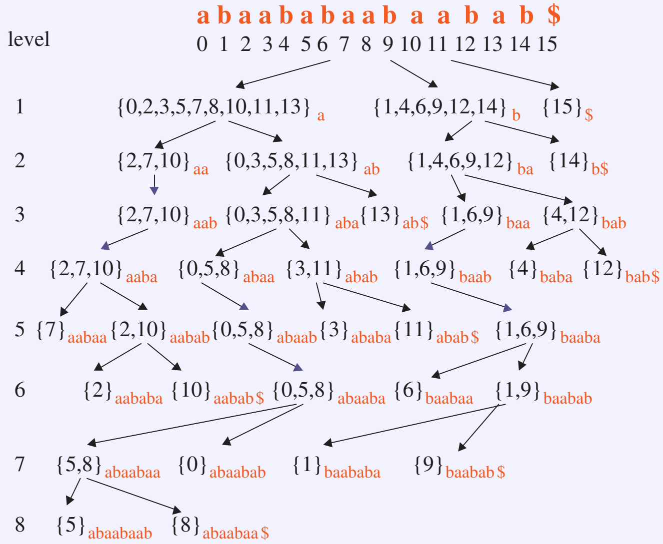
The process of refinement and update of the gap function is kept to $O(n \log n)$ complexity.

The algorithm, *cont.*

The maximal repetitions are determined from the information in the gap function once a level is completed.

The algorithm, *cont.*

Partitioning-based Runs Algorithm
Motivation
Background
Algorithm
Conclusion



The algorithm, *cont.*

Once a level is completed, the gap function contains the following information:

$Gap[p] = e$ iff e has an immediate predecessor $\hat{e}$ in the same class and $e - \hat{e} = p$.

The algorithm, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

Thus if processing level p, every entry in $Gap[p]$ determines a primitevely rooted square in the input string.

For instance consider the class $\{0, 3, 5, 8, 11, 13\}$ on level 2 capturing the occurrences of ab. It follows that there are square $abab$ starting at positions 3 and 11.

The algorithm, *cont.*

Since every run consists of a "bunch" of primitevely rooted squares, we can "consolidate" all the squares into runs. The problem is that there is no discernible order of the entries in the gap list, hence we get the squares in an arbitrary order if we just follow the gap list.

The algorithm, *cont.*

We instead test in constant time if a square can be “shifted” one position to the left and/or one position to the right. If so, we continue, but we must use auxialiary data structures (the ones used for refinemt but idle at this point) to “remember” that we already used a particular square in order not to use it when we come to it in the gap list.

The algorithm, *cont.*

The tracing of maximal repetitions is as before, while for counting of distinct squares, only the first square of each class is counted.

The control of what should be computed, whether

runs, repetitions, or distinct squares is via compilation flags.

Conclusion

We presented an implementation of Crochemore's partitioning algorithm based algorithm to compute runs, maximal repetitions, and primitively rooted distinct squares. The algorithm is reasonably memory efficient ($13n$ inetegers) and reasonably fast (empirical observation).

Conclusion, *cont.*

Partitioning-based Runs Algorithm

Motivation
Background
Algorithm
Conclusion

A true validation of how fast it is is missing. A graduate student of Smyth and myself is working on comparison with Chen+Puglisi+Smyth 2007 algorithm (reputedly the fastest) and with Hirashima+Bannai+Matsubara+Ishino+Shinohara (reported at PSC in 2009) bit-parallel implementation.

Conclusion, *cont.*

Another graduate student of mine is working on a `C/C++` implementation of a linear runs algorithm based on a Java implementation by Johannes Fischer. This will also be used for comparison measurements.

Conclusion, *cont.*

We would welcome any `C/C++` implementation of runs algorithm for further comparisons.

Thank you