**Response to Review Comments**

**Reviewer #1:**

**Comments:**

The paper is well written up to Section 2.  In contrast, Section 3 requires readers to have prior knowledge about induced suffix sorting algorithm, and in the current format, it is nearly impossible for readers without background to understand the details, not to mention how to check the correctness.

While I believe that the results in Section 3 are correct, but with the current writing, it is hard for me to verify its correctness.  A major revision is needed, most suitably by adding enough examples, and perhaps a brief introduction to induced sorting as well.

**Response:**

We have rewritten the whole paper to make it easy to follow. Besides, an example is provided to explain the basic idea of the induced sorting principle in the end of the paper for reference.

**Comments:**

Your proposed methods require sort(n) I/Os to perform.  Theoretically speaking, the fastest suffix array construction method and LCP method also require sort(n) I/Os.  Am I correct? If this is the case, what is the benefit of applying your method, instead of implementing the above methods (or, asking an independent programmer to implement these methods if you are using that already) and compare the results? To my understanding, a checker should take much less time or I/Os than a brute-force re-implementation.  Please comment the above in your revision.

**Response:**

We evaluate the performance of our checking algorithms in comparison with that of the two solutions composed of current fastest sequential and parallel algorithms for suffix and LCP arrays construction. The results indicate that, our programs for the algorithms designed by Method A and B outperform that for the sequential solution on various real-world datasets, but their speeds are half as that for the parallel solution under the given conditions.

We point out that our checking algorithms can achieve optimal time and I/O complexities in external memory, but both the sequential and the parallel solutions take time and I/O overheads proportional to n^2/M. Hence, they will suffer from a limitation to the scalability when the input string length n is much bigger than the internal memory capacity M. This was also reported in the experimental results in references [23-24]. Actually, our programs may have a substantial improvement by parallelizing their sorting and inducing processes, but their implementation design will be also complicated as well. From our perspective, a checking algorithm should be not only fast but also versatile. In this aspect, our first method can be generalized to check the lexical order and/or the LCP-value for one or multiple pairs of suffixes simultaneously, where each suffix is allowed to start at any position of the input string. This feature makes it possible to trivially adapt Algorithm 1 for verifying a sparse/finite-order SA and the corresponding LCP array.

**Reviewer #2:**

**Comments:**

The paper seems to be well-written. But I have a serious problem with the motivation of this work. It is to check suffix trees and LCP arrays produced by some existing algorithms in case of implementation bugs or occational errors. The Karp-Rabin fingerpring function is used to do the task.  
My concern is, the implementation bugs and occational errors should be checked by testing codes, not by running another program since the program itself may have implemtation bugs or occational errors. So we have to run a third program to check the second. Finally, we will end up with examining the codes of some algorithm.  
The program verification can be useful. But the paper does not fall in this categary. It does not contribute anything to improving the efficient construction of suffix trees and LCP arrays. Given this, I find it very difficult to support the accepatance of the paper.

**Response:**

A program for constructing a suffix/LCP array gives no guarantee that it has correctly implemented the underlying algorithm. As a common practice, a suffix/LCP checker is provided to check the correctness of a constructed array. For example, such a checker can be found in some widespread software packets, such as SA-IS, eSAIS, DC3 and so on.   
A checker is also demanded for an array constructed by a probabilistic algorithm. In this case, the array is correctly constructed with a probability and hence must be verified by a checker to ensure its correctness. To the end, we can check the result by comparing it with the one produced by another builder. But this is not feasible in all the cases. For example, we cannot directly use the output of an infinite-order SA builder to verify a finite-order SA and vice versa. This motivates us to design algorithms/methods specific for the checking purpose. As a result, our first checking method can be used to check the correctness of the lexical order and the LCP-value for one or multiple pairs of suffixes starting at any positions of the input within optimal time and I/O complexity. Actually, this method can be also applied to many other applications, such as error detection and data recovery.

**Reviewer #3:**

**Comments:**

The only thing I miss throughout the manuscript is an appendix where the checking methods could be illustrated by means of an example. That is, to exemplify the different detailed steps and main structures involved into each proposal, by using a sample input string. I would really appreciate that authors could include that section as, in my opinion, it would help to enhance the paper's content even more. Besides that, and just as a small detail, it would also be interesting that authors could point out the specific opportunities they find useful to improve their contributions in the near future, at the end of the conclusions.

**Response:**

In the revised paper, we have redesigned the second checking method for better I/O and space efficiencies. We give the framework of the algorithms designed by the proposed checking methods and describe each step with details for a better understanding. For readers unfamiliar with the induced sorting principle, we present in the end of the article its basic idea with an example of the inducing process of IS-based construction algorithms. To make a performance analysis, we compare the programs for the proposed algorithms to that for the current fastest sequential and parallel construction algorithms in terms of running time and show potential ways of improving their performance as well.

Thanks a lot!