**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 rewritten the whole paper to make it easier to follow. Particularly, two examples are provided for demonstrating the basic ideas of the induced sorting principle and the fingerprinting technique.

**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 make a performance comparison between the proposed checking algorithms and the selected construction algorithms. It can be observed that the programs for the checking algorithms outperform that for the current fastest sequential construction solution on the real-world datasets, but their speed are half as that for the parallel solution under the given conditions. We point out that Algorithms 1-2 have better time and I/O complexities than the parallel solutions, where the former are within sorting complexity and the latter are proportional to n^2/M. Therefore, it is more reasonable to compare the results of our programs with that of the sequential solution for big n.

We emphasize that our current programs are for experimental study only, and there is still a big margin for better implementations. For example, a recent work [20] for engineering the IS methods achieved a significant improvement over the previous results [10, 12, 13]. This indicates a great potential for speeding up ProgB/ProgB+, because the induced sorting process is the performance bottleneck for Method B.

From our perspective, a checking algorithm should be not only fast but also general. In terms of this aspect, Method A can be applied to verifying any (sparse) suffix/LCP array of any order, while the existing construction algorithms are not capable of checking their outputs mutually in some cases.

**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 for program debugging, we usually check the output of a (probabilistic) builder by comparing it with that of another (deterministic) builder. However, this approach is not feasible in all the cases and thus motivates us to design general method specific for the checking purpose. As a result, Method A is of high generality, it is capable of verifing any (sparse) suffix/LCP arrays of any order within sorting time and I/O complexity and applicable for 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:**

We make the following modifications to improve the readability of our paper:

1. The framework of the algorithms designed by the proposed checking methods are demonstrated in Sections 2-3 with each step explained in detail.
2. An illustrated example for describing the inducing process of SA-IS is given in the appendix for readers unfamiliar with the induced sorting principle.
3. A performance comparison between Algorithms 1-2 and the current fastest construction algorithms are given in Section 4, together with a discussion on the improvements of implementation and design.

From our study, the proposed checking methods is more scalable than the state-of-the-art construction algorithms in the following two aspects:

1. The algorithms for Methods A and B are within sorting complexity, while the current fastest construction algorithms have time and I/O complexities proportional to n^2/M.
2. Method A is able to verify any (sparse) suffix and LCP arrays of any order, while the existing construction algorithms are not.

Thanks a lot!