Enclosure-1

Exploring the Hierarchical Overlap Graph for the All Pairs Prefix Suffix Problem

A LAB PROJECT REPORT

Submitted by

Akula Pradeep Kumar (20114004)

Alapati Sri Vinay(20114005)

Nalla Jagannath Reddy(20114060)

Under the guidance of

Prof. Shahbaz Khan

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

ROORKEE-247667

April 2023

Enclosure-2

CANDIDATE'S DECLARATION

I declare that the work carried out in this report entitled " **Exploring the Hierarchical Overlap Graph for the All Pairs Prefix Suffix Problem** " is presented on behalf of the fulfillment of the course CSN-300 submitted to the **Department of Computer Science and Engineering**, **Indian Institute of Technology Roorkee** under the supervision and guidance of **Prof. Shahbaz Khan , CSE Dept..**

I further certify that the work presented in this report has not been submitted anywhere for any kind of certification or award of any other degree/diploma.

Date: 4th , May 2023

Place: Roorkee

Signature

Akula Pradeep Kumar(20114004)

Alapati Sri Vinay(20114005)

Nalla Jagannath Reddy(20114060)

Enclosure-3

CERTIFICATE

This is to certify that the above statement made by the candidates is correct to the best of my knowledge and belief.

Date:

Place:

(Signature of the Supervisor)

Introduction

The All Pairs Suffix Prefix Problem is a well-known problem in computer science, with applications in fields such as genomics and bioinformatics. Given a set of strings, the problem asks for all pairs of strings that share a common suffix and prefix

.

The All Pairs Suffix Prefix Problem is a challenging problem that requires a deep understanding of various data structures and algorithms. Studying the problem can help researchers develop more efficient algorithms for searching and analyzing large datasets, which is becoming increasingly important in fields such as genomics and bioinformatics. Additionally, the problem provides a useful test case for evaluating the performance of different algorithms and data structures.

Theoretical Background

The All Pairs Suffix Prefix Problem can be solved using a variety of algorithms, including the Trie-based algorithm and the Aho-Corasick algorithm. The Trie-based algorithm involves building a Trie data structure from the set of strings and then traversing the Trie to identify all pairs of strings that share a common suffix and prefix. The Aho-Corasick algorithm is a more efficient alternative that involves building a finite automaton to search for the common prefixes and suffixes.

Another approach to solving the problem is to use the Hierarchical Overlap Graph (HOG), which is a graph-based representation of the set of strings that allows for efficient identification of common prefixes and suffixes.

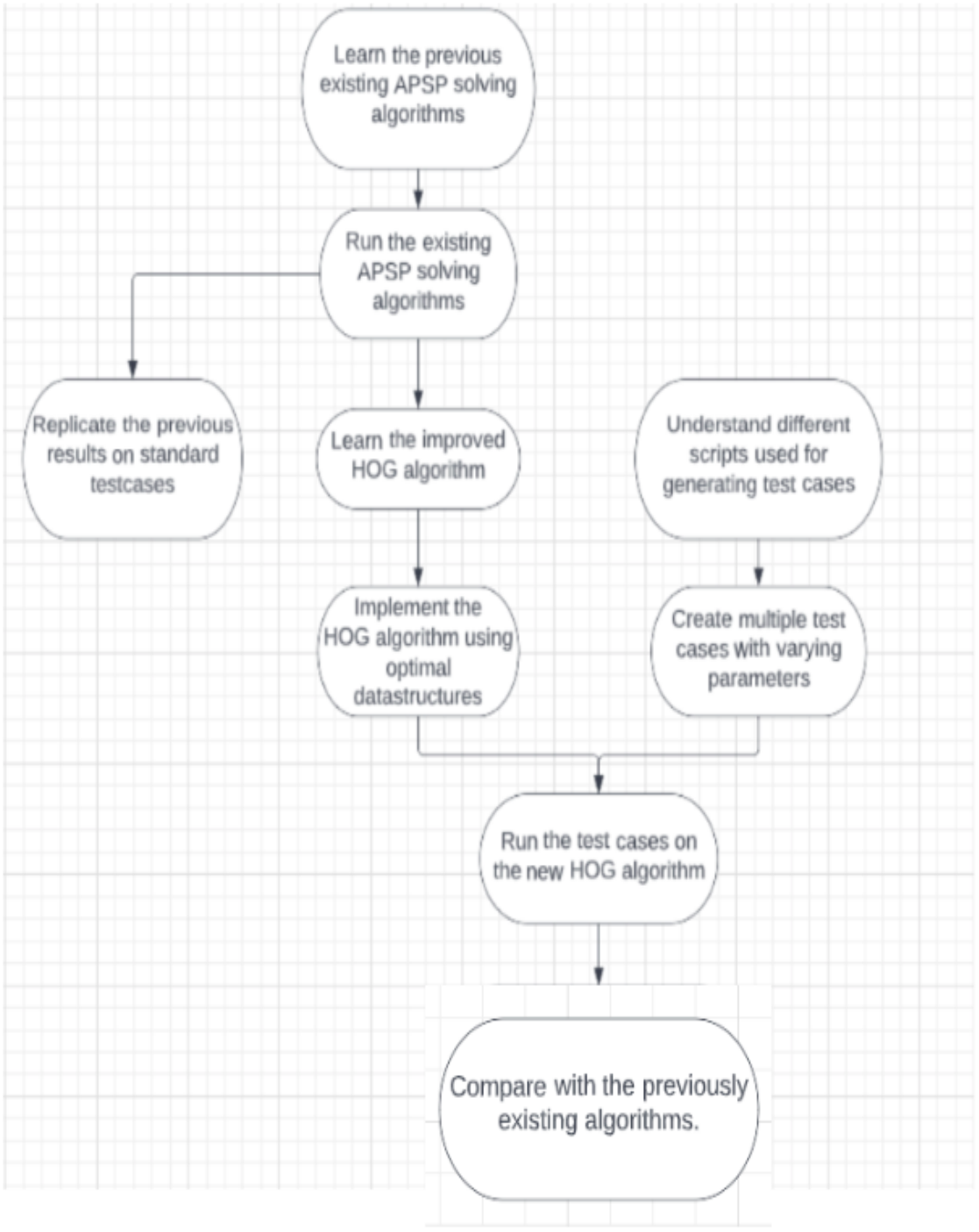
Also HOG only reports a set of overlaps while the other algorithms report all pairs of overlaps.

Literature Review

- Gusfield et al. (Gusfield et al., 1992) presented an optimal solution for APSP using a generalized suffix tree (GST). A suffix tree of a text T is a data structure in which each suffix in T is represented by a path from the root to a leaf.
- The high space-consumption of suffix trees motivated Ohlebusch and Gog(Ohlebusch and Gog, 2010) to present a practically better solution in terms of time and space using a generalized enhanced suffix array (GESA). An ESA is a suffix array and an LCP (longest common prefix) array. A Suffix array of a text T with a size of n is an array of size n containing the text positions of lexicographically sorted suffixes of T. An LCP array is an array of size n containing the lengths of the longest common prefixes of every two consecutive lexicographically sorted suffixes of T. The word ”Generalized” is used to indicate that the data structure is created from one string that is built by concatenating all reads (sequences) together and separating every two reads with a distinct separator.
- The usage of compressed versions of these data structures in some works such as (Simpson and Durbin, 2012), (Haj Rachid et al., 2014b) and (Haj Rachid et al., 2014a) offered low space consumption solutions for APSP, however, it had a dramatic slowdown effect. Despite the optimal time complexity of GST and ESA, it has been realized that these solutions have high constants when solving APSP.

- Accordingly, practical solutions such as Readjoiner (Gonnella and Kurtz, 2012) and SOF (Haj Rachid and Malluhi, 2015) are practically faster solutions and consume much less space than GST and GESA.•Tustumi et al (Tustumi et al., 2016) revisited the OG solution using ESA and improved the time (2.6 times faster) and the space (15% less) consumptions. The work of (Louza et al., 2016) presented a technique to parallelize ESA’s new technique. The presented parallelization achieved superior speed–up over the sequential version, however, Readjoiner and SOF have better time and spaceconsumptions.•Lim and Park (Lim and Park, 2017) recently presented a solution which uses the same data structure that is utilized in SOF in addition to other few auxiliary data structures. The presented solution uses advanced algorithmic techniques for the matching step in order to achieve fast running time. The time consumption is dramatically improved over SOF with an expected higher space-consumption than SOF. They called their algorithm FastAPSP

Methodology



HOG Algorithm

De Bruijn graphs are created by using k-length substrings (k-mers) of reads as nodes and arcs denoting k-1 length overlaps among the k-mers. They have the advantage of being linear in size, but they lose information about the relationship of k-mers with the reads and cannot represent overlaps of size other than k − 1. Overlap graphs, on the other hand, have each read as a node, and edges represent maximum overlap between nodes. They store more overlap information than De Bruijn graphs but are inherently quadratic in size and do not maintain whether two pairs of strings have the same overlap. Hierarchical Overlap Graphs are an alternative that can overcome these limitations.

Hierarchical Overlap Graphs (HOG) were proposed as an alternative to the limitations of de Bruijn and overlap graphs. HOG has nodes for all the longest overlaps between every pair of strings, and edges connecting strings to their suffix and prefix, using linear space. HOG exploits pairs of strings having the same longest overlaps, requiring linear size instead of quadratic size, which Overlap graphs require. Therefore, HOG is a promising alternative to both de Bruijn and Overlap graphs to better solve the problem of genome assembly. It also has the potential to better solve the approximate shortest superstring problem having applications in both genome assembly and data compression.

The Hierarchical Overlap Graph (HOG) is a graph structure used in genome assembly, and it is created using the Aho-Corasick trie and functions such as MarkH. The algorithm for MarkH has been improved in a recent paper ([2102.02873] Optimal Construction of Hierarchical Overlap Graphs).

The HOG contains nodes representing the longest overlaps between pairs of strings, and edges connecting strings to their prefixes and suffixes. This graph structure offers a promising alternative to both de Bruijn and overlap graphs for solving the problem of genome assembly. By maintaining information about whether two pairs of strings have the same overlap, the HOG also has the potential to better solve the approximate shortest superstring problem, which has applications in both genome assembly and data compression.

The implementation can be found at :
https://github.com/Jagannath1/HOG/tree/main

Experimental Setup

We are comparing algorithms :

- HOG
- FastAPSP
- SOF
- ReadJoiner

where we are not producing any output , and also minimum overlap is also set to 1 in all the cases, we are also operating all the algorithms on a single thread. We have generated random test cases using the gen function in SOF directory. We have run these algorithms on a machine running Linux Ubuntu with 2GHz CPU 64 GB RAM.

We have written bash scripts to automatically run multiple test cases on all the 4 algorithms. These test cases are run serially on all 4 algorithms.

Test Cases

We have generated random test cases with the following parameters :

At constant k=1000 , N varies from 10^6 to 2*10^7.

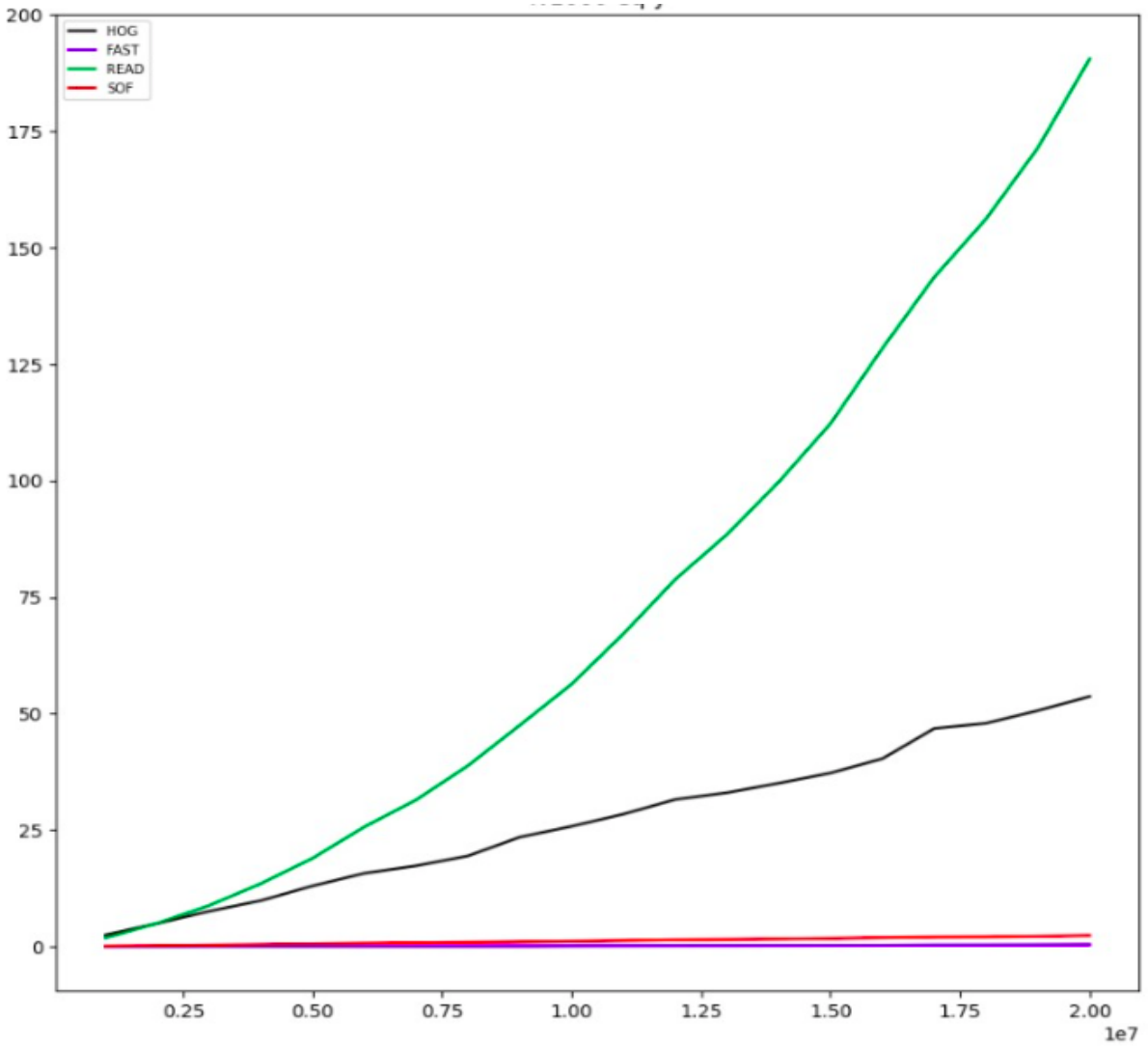
At constant N=10^6 , k varies from 5000 to 10^5.

We have generated 20 test cases for each set of values of N and K and calculated the average time taken by each algorithm. This nullifies any error present in the calculation of time taken by the algorithm.

Results

We have written python code to plot graphs and here are the results for:

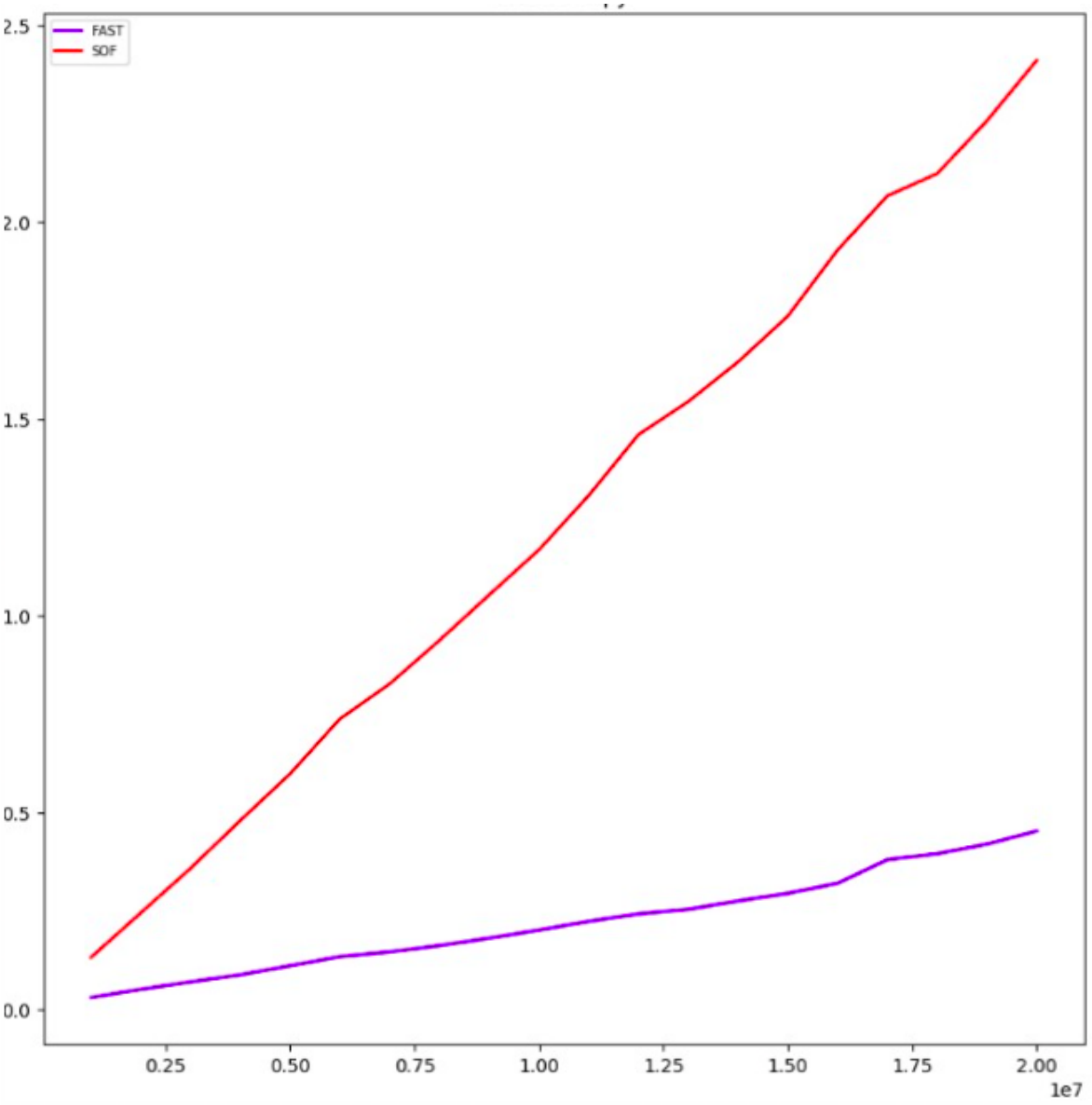
k=1000 and N varies :



Here we can see that for HOG we get a linear Graph and for ReadJoiner we can see that it is proportional to n^2.

Similarly The theoretical time complexity of HOG also checks out as it is O(n).

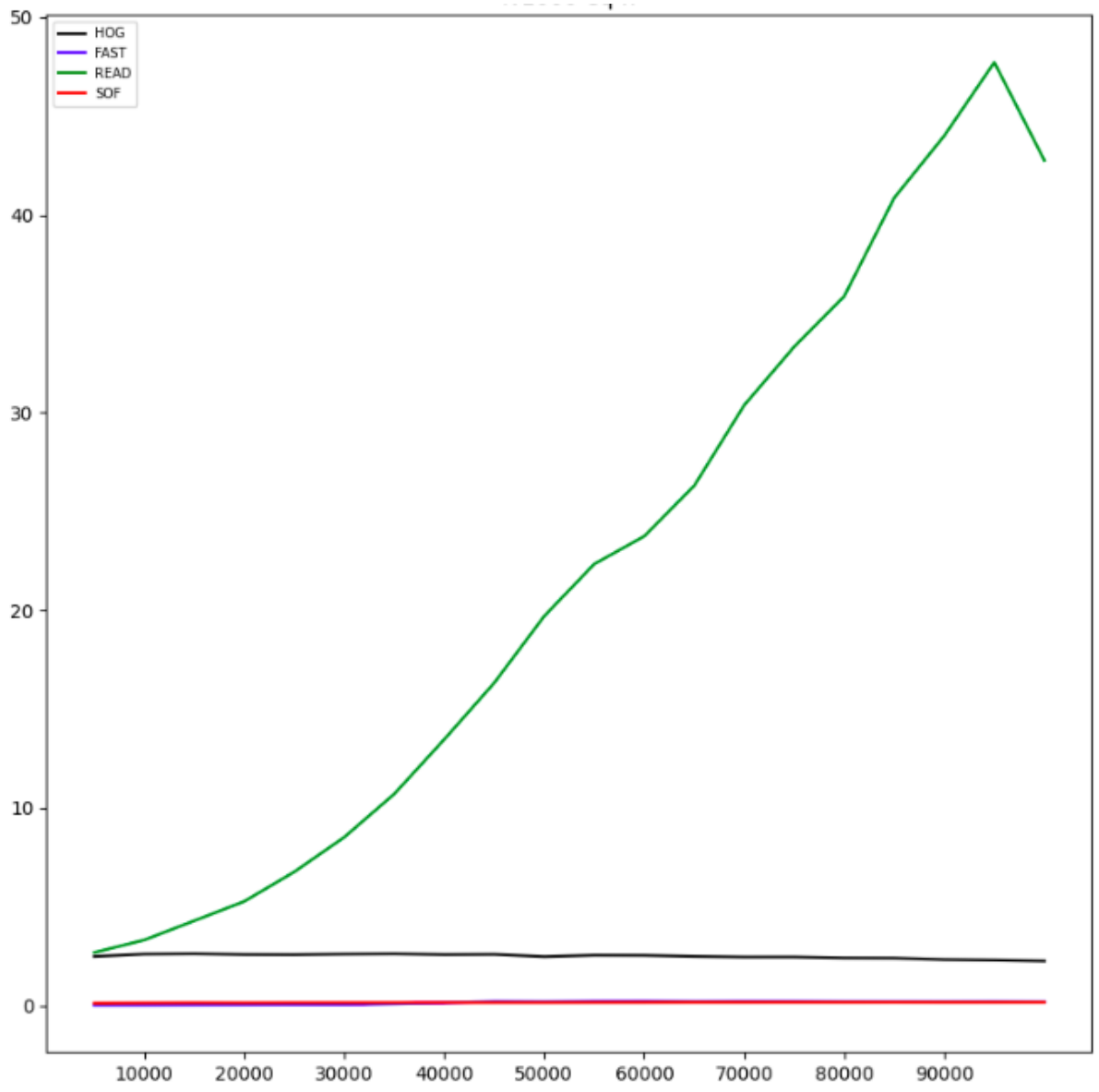
In the case of SOF and FastAPSP it may look like they are constant but if we zoom in we get :



As we can see these are also linear .

The theoretical time complexity of FastAPSP and SOF also checks out as they are proportional to O(n) when K is constant.

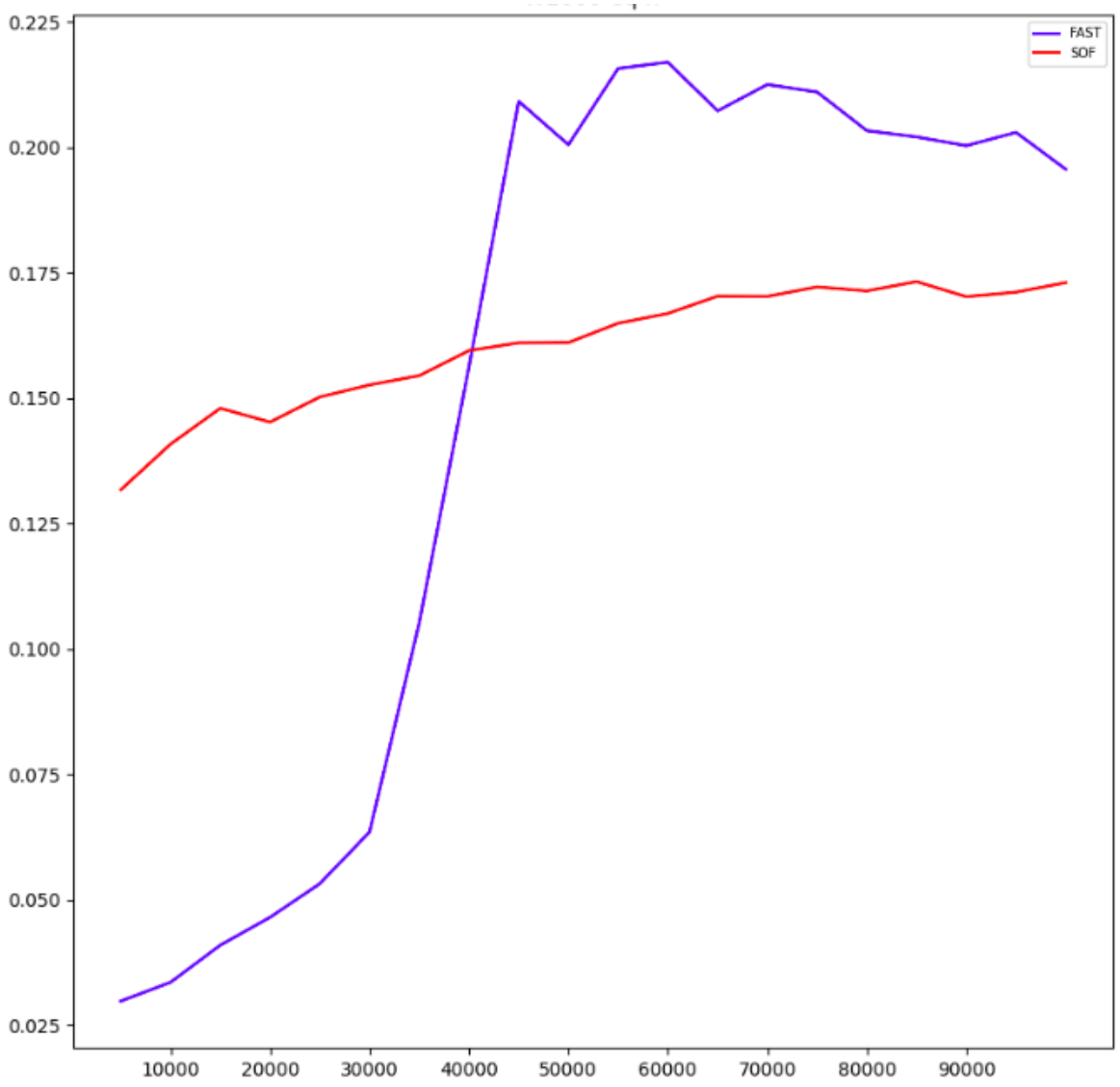
For N= 10^6 and K varies



Here we can see that ReadJoiner linearly depends on the value of k.

In this case we can see that the values of Readjoiner are too high compared to other Algorithms.

In the case of SOF and FastAPSP if we zoom in we get :



The sudden jump in the graph of FastAPSP could be explained as the value of K increases : The algorithm constructs a suffix-prefix matrix for the set of strings. The suffix-prefix matrix is a square matrix with dimensions equal to the number of strings in the input set. This could have exceeded the memory limit and thus taking more time .

Conclusion

- As expected the times are increasing with n in all the cases although it may seem that SOF and fast-APSP are running in constant time.
- But when you exclusively plot these two algorithms you can see it is also linear but the constant is two low compared to the other two algorithms as shown in the 2nd graph.
- HOG is running in linear time from the 1st graph.
- HOG is not showing any dependency on k from the 3rd graph.
- HOG is performing better than ReadJoiner
- Both are slow compared to SOF and Fast-APSP.

Although the implementation of HOG looks good in performance it may be resulting in huge Overhead. So, we should work on eliminating that if possible.

References

“*″*.” *″* - *Wiktionary*, https://github.com/maanrachid/SOF. Accessed 4 May 2023.

“*″*.” *″* - *Wiktionary*, https://github.com/maanrachid/AOF. Accessed 4 May 2023.

“Maan Haj Rachid (0000-0002-6380-209X).” *ORCID*, 31 May 2022, https://orcid.org/0000-0002-6380-209X. Accessed 4 May 2023.

“ALuesink/APSP_algorithm.” *GitHub*, https://github.com/ALuesink/APSP_algorithm. Accessed 4 May 2023.

“Index of /pub.” *GenomeTools*, http://genometools.org/pub/. Accessed 4 May 2023.

Khan, Shahbaz. “[2102.02873] Optimal Construction of Hierarchical Overlap Graphs.” *arXiv*, 4 February 2021, https://arxiv.org/abs/2102.02873. Accessed 4 May 2023.

"Publication Details." *SciTePress*, https://www.scitepress.org/Link.aspx?doi=10.5220/0007369801740181. Accessed 4 May 2023.

"SNUCSE-CTA/FastAPSP: A fast algorithm for the all-pairs suffix–prefix problem." *GitHub*, https://github.com/SNUCSE-CTA/FastAPSP. Accessed 4 May 2023.