Persistency in Suffix Trees with Applications to String Interval Problems

Tsvi Kopelowitz[1], Moshe Lewenstein[2,⋆], and Ely Porat[2,⋆⋆]

[1] Weizmann Institute of Science Rehovot, Israel
[2] Bar-Ilan University, Ramat-Gan, Israel

Abstract. The suffix tree has proven to be an invaluable indexing data structure, which is widely used as a building block in many applications. We study the problem of making a suffix tree persistent. Specifically, consider a streamed text T where characters are prepended to the beginning of the text. The suffix tree is updated for each character prepended. We wish to allow access to any previous version of the suffix tree. While it is possible to support basic persistence for suffix trees using classical persistence techniques, some applications which can make use of this persistency cannot be solved efficiently using these techniques alone.

A collection of such problems is that of queries on string intervals of the text indexed by the suffix tree. In other words, if the text $T = t_1...t_n$ is indexed, one may want to answer different queries on string intervals, $t_i...t_j$, of the text. These types of problems are known as position-restricted and contain querying, reporting, rank, selection etc. Persistency can be utilized to obtain solutions for these problems on prefixes of the text, by solving these problems on previous versions of the suffix tree. However, for substrings it is not sufficient to use the standard persistency.

We propose more sophisticated persistent techniques which yield solutions for position-restricted querying, reporting, rank, and selection problems.

1 Introduction

Text indexing is one of the most important paradigms in searching. The idea is to preprocess a text $T = t_1 \cdots t_n$ over alphabet Σ and construct a mechanism that will later provide answers to queries of the form "report all of the occurrences of a pattern P in the text" in time proportional to the size of the *pattern* and output, rather than the size of the text. The suffix tree [10,14,16,17] has proven to be an invaluable data structure for indexing. It is also considered a building block for various other indexing and non-indexing problems.

Some of the suffix tree constructions work in the *online* model [16,17], in which one maintains a suffix tree for a text that arrives character by character, and at any given time we might receive a pattern query. For simplicity, we assume that

⋆ This research was partially supported by the ISF (grant no. 1848/04).
⋆⋆ This research was partially supported by the BSF (grant no. 2006334) and the ISF grant (grant no. 1484/08).

R. Grossi et al. (Eds.): SPIRE 2011, LNCS 7024, pp. 67–80, 2011.

the text arrives from the end towards the beginning. Otherwise a single character added at the end of the text can impose a linear number of changes to the suffix tree. Of course, if the text arrives from beginning to end we can view the text in reversed form and then a queried pattern is reversed as well in order to obtain the correct results. The best currently known results for the online suffix tree are an $O(\log |\Sigma|)$ amortized time per character by Weiner in [17], and $O(\log n)$ worst case per character by Amir et.al in [2]. We also note that for constant-size alphabets there is a different indexing structure by Amir and Nor [3].

Data structures which have the ability to allow access to previous versions of themselves over the updates are known as *persistent data structures* [5,8,9]. For a good survey see [12]. We focus on two types of persistent data structures. The first type is *fully persistent data structures*, in which an update can be made to any version of the data structure. In this type, one can imagine a tree of versions of the data structure as update operations are performed on various versions. The second type is known as *partially persistent data structures*, in which an update can be made only to the last version created. In this type, one can imagine a list of versions of the data structure as update operations are performed only on the tail of the list. In Section 2 we will provide a quick review on some of the known results in this field which we will later use.

To the best of our knowledge persistent suffix trees have not been considered before. Nevertheless, since suffix trees have constant indegree it follows that one can make suffix trees persistent using the result of [8]. However, this persistency is useful solely for navigation purposes, which is sufficient for various standard applications, e.g. queries of the sort "report all of the times in which a specific stock has a series of consecutive values in the stock market, before last March". More sophisticated queries cannot be answered with navigational data on the current text alone.

One subset of problems that we focus on is string interval problems, a.k.a. position restricted problems. Here one has a suffix tree for the full text $T = t_1 \cdots t_n$ but is interested in queries that are narrowed down onto an interval $t_i \cdots t_j$. One problem is known as position restricted indexing, see [13], position restricted reporting, where one desires to report all matches within an interval of the text, position restricted rank, where one desires to know the rank of a given pattern within the interval of the text, and position restricted select, where one desires to find the i'th appearance of a given pattern. The intuition for using a persistent suffix tree for these type of problems is that by accessing the version of the suffix tree just after t_i was added, one may reduce the problem to searching within the prefix of $t_i \cdots t_n$ of length $j - i + 1$.

Unfortunately, the persistent suffix tree on its own does not suffice for efficient solutions for these problems. This happens because versions of the data structure provide bounds for one side of the desired interval query, but not both. Hence, we need to provide a persistent mechanism which supplies the capability for answering the different queries for position restricted problems. We do this by providing a general framework solution, and then show how each of the above applications can be solved using this general framework.