Distributed and Paged Suffix Trees for Large Genetic Databases

Raphaël Clifford and Marek Sergot

raphael@clifford.net, m.sergot@ic.ac.uk

Imperial College London, UK

Overview

- Why do we need distributed and paged suffix trees?
- What do we mean by a distributed or paged suffix tree?
- How can we construct one efficiently?
- What can we do with it?

Motivation

Why do we need distributed and paged suffix trees?

- Suffix trees use a lot of memory.
- Their construction and querying algorithms have very poor memory locality so secondary storage can not be used.
- Sequencing projects are producing more and more data.
- Smarter bioinformatics algorithms are being developed which require more complex data structures such as the suffix tree.

Sparse suffix trees

A sparse suffix tree (SST) of a string t is the compacted trie of a subset of the suffixes of t.

- Evenly spaced suffixes handled by Karkkainen and Ukkonen in 1996.
- Andersson gave a quite different construction algorithm for “suffix trees on words”. The suffixes that are to be inserted into the tree are defined by delimiters that define the end of “words” in the string.
- Andersson’s construction algorithm is complicated but is able to cover the case of evenly spaced suffixes as well as some others. However it is unnecessary as we shall see.

SSTs for distributed computing

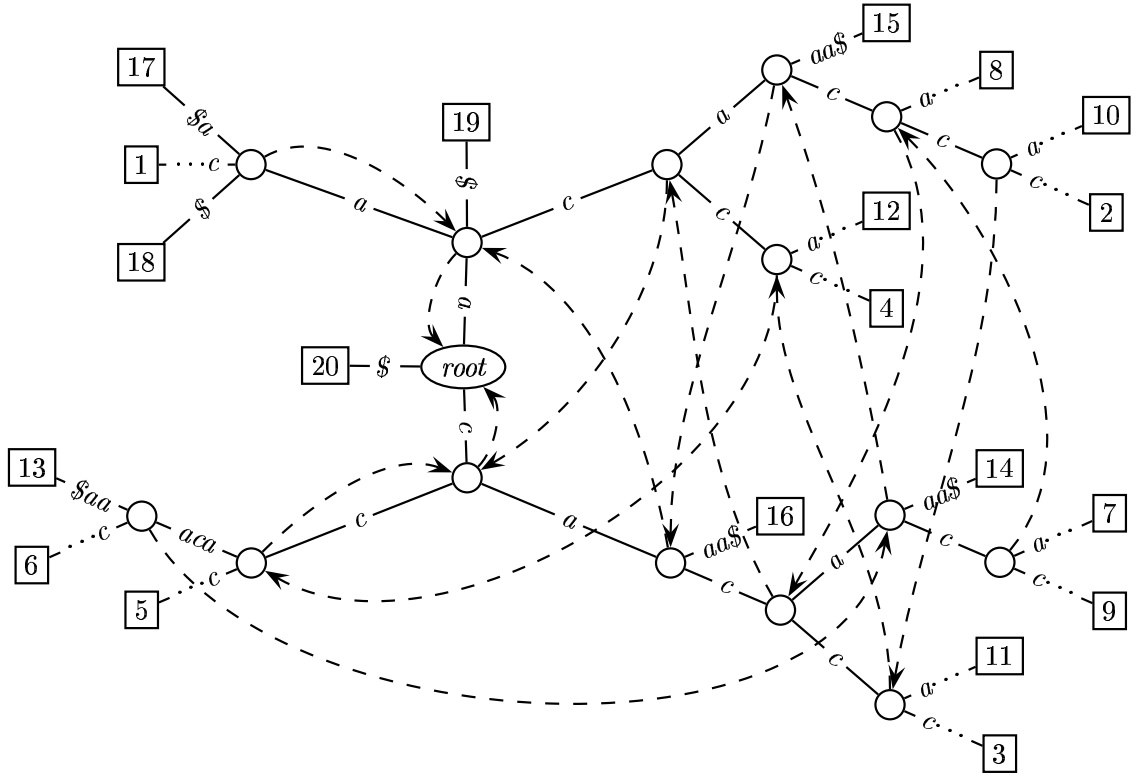
- We distribute the suffix tree by splitting the text *lexicographically.*
- Define the suffixes that are to inserted by referring to a set of *valid* suffixes, V_z, where z is a short fixed substring of the text. V_z contains the start positions of each suffix that has z as a prefix. We say that a substring of string t is *nodal* wrt to a SST if it corresponds to a node in that SST.
- A distributed suffix tree (DST) or paged suffix tree is simply a collection of SSTs.

Sparse suffix links

- Suffix links will in general point across SSTs defined in this way. To perform the construction efficiently we must redefine them
- We require that sparse suffix links must point to nodes within the same SST and that they are powerful enough to enable linear time construction.

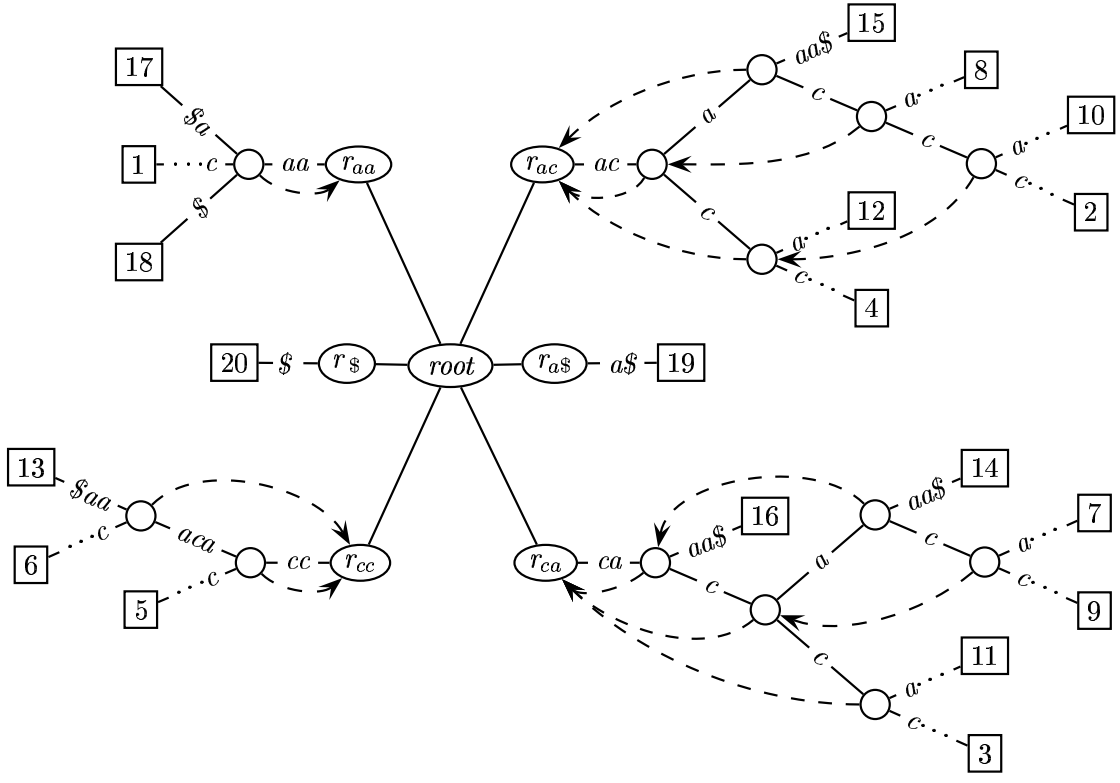
Definition:
Consider an input pair (t, V_z) and sparse suffix tree $T = sst(t, V_z)$. Let aw be nodal in T and v be the longest repeated suffix of aw that occurs in T. A *sparse suffix link* or *ssl* is an unlabelled edge from $\overline{aw}$ to the *root* if $|v| < |z|$ and from $\overline{aw}$ to $\overline{v}$, otherwise.

Normal suffix tree



The standard suffix tree of $aacacccacacaccacaaa\$$ with standard suffix links.

Merged SSTs



SSTs for $aacacccacacaccacaaa\$$ with their respective root nodes. The sparse suffix links are marked with dashed arrows

SST construction

In order to construct the SST online we need to be sure which new suffixes might need to be inserted at each turn.

Definition:

Consider input pair (p, V_z). Denote the set of valid repeated suffixes of p by $R(p, V_z)$. We define this set to include the empty string, ϵ. Let $\alpha(p)$ be the longest suffix in $R(p, V_z)$.

Example:

Consider input string $t = aabaaa$ with prefix $p = aabaa$ and $V_{aa} = \{1, 4, 5\}$. $\alpha(p) = aa$ and $R(p, V_{aa}) = \{aa, \epsilon\}$.

SST construction (contd.)

Theorem 1 *Consider the input pair (t, V_z) with p, a proper prefix of t. Let a be the character in t that directly follows the prefix p. Consider also the set, S, of valid suffixes, sa, for $I[1, |pa|]$ such that $|\alpha(p) \geq |sa| > |\alpha(pa)|$ and $s \in R(p, V_z)$.* $sst(pa, V_z) = sst(p, V_z)$ augmented by S.

Proof outline

Proof 1 *Let sa be a valid suffix for $I[1, |pa|]$. We need to insert this new suffix into the tree if and only if $s \in R(p, V_z)$ but $sa \notin R(pa, V_z)$ ($s = \epsilon$ is a special case). In this case sa corresponds to a leaf in $sst(pa, V_z)$ but s does not correspond to a leaf in $sst(p, V_z)$ so a new node is necessary.*

If $|sa| > |\alpha(p)a|$then both s and sa will correspond to the same leaf in $sst(pa, V_z)$.

If $|\alpha(p)a| \geq |sa| > |\alpha(pa)|$then $sa \notin R(pa, V_z)$.As $\alpha(p) \geq |s|$we know that $s \in R(p, V_z)$ and therefore a new leaf is needed.

If $|\alpha(pa)| \geq |sa|$ then either $sa \in R(pa, V_z)$or $|sa| = a$. In neither case is a new leaf required (as $|\alpha(pa)|$is of non-zero length).

Algorithm

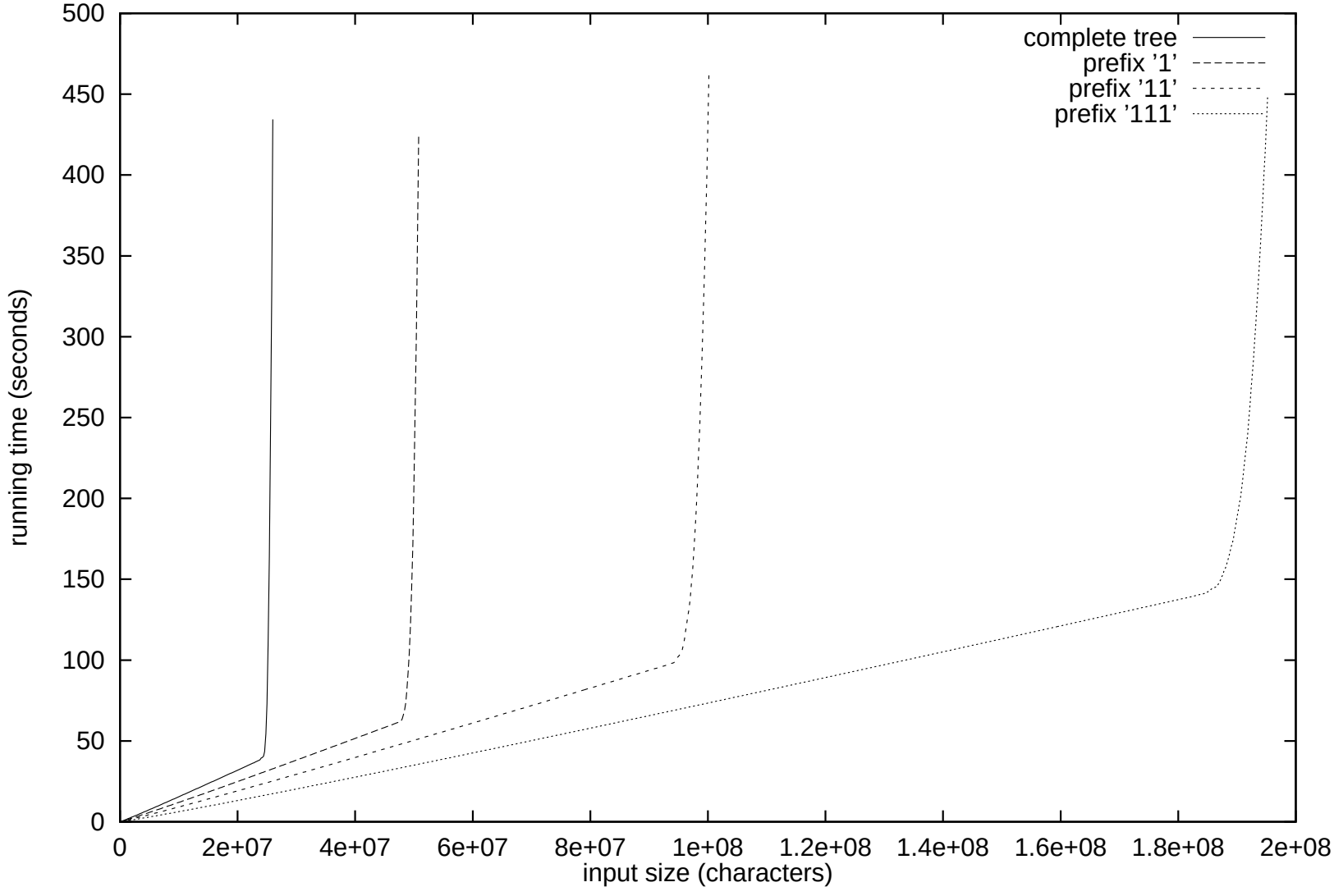
There are two main differences to Ukkonen's standard suffix tree construction algorithm,

- *ssl's* used instead of standard suffix links
- If an *ssl* is followed to the root we must skip along the text to the next valid suffix of the current prefix and ensure we are at the relevant position in the tree.

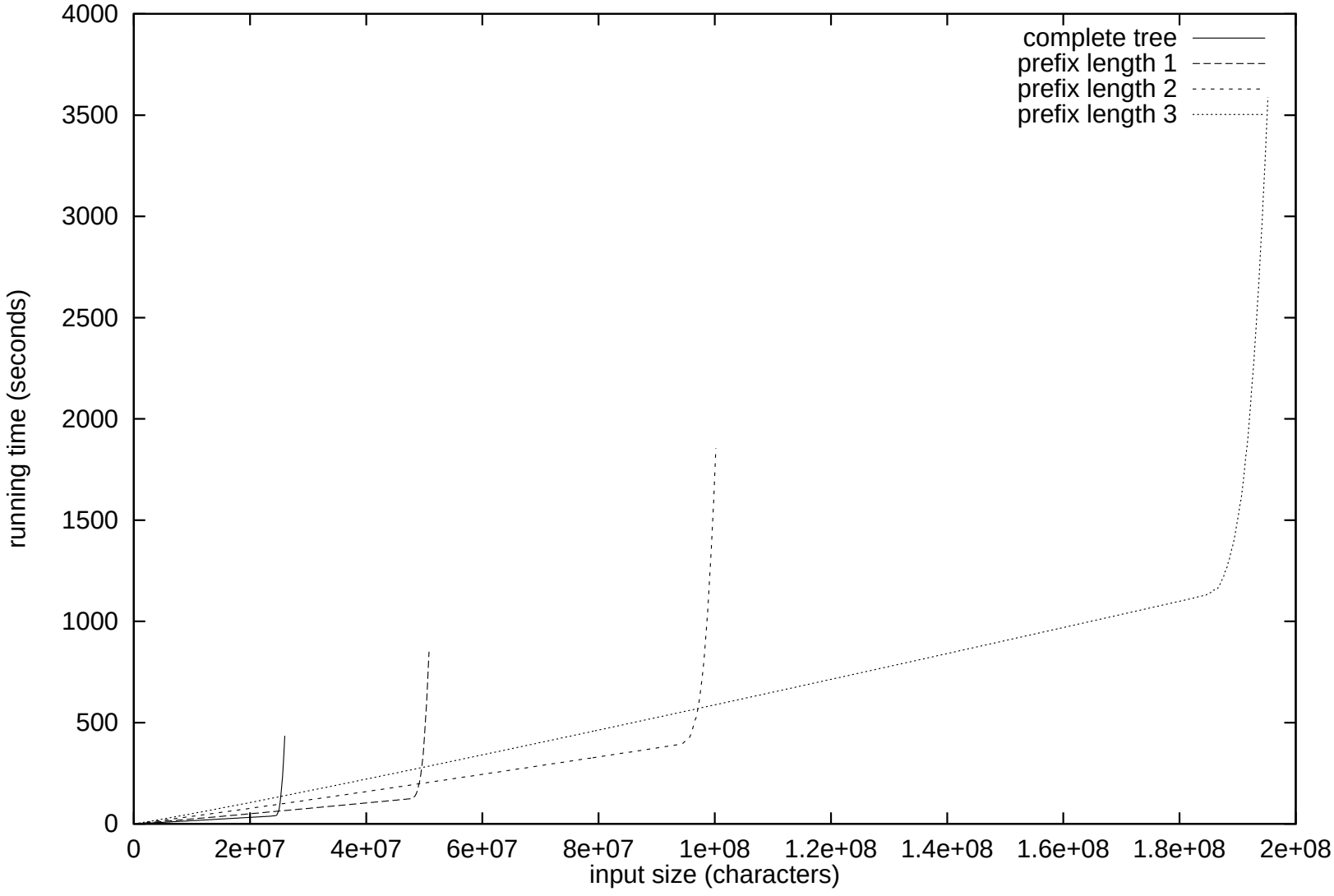
Using exactly the same algorithm Andersson's suffix trees on words and hence evenly spaced SSTs can also be constructed.

Theorem 2 *For an input pair* (t, V_z) *with* $|z| \geq 1$ *and constant,* $sst(t, V_z)$ *can be constructed in* $O(n)$ *time and* $O(V_z)$ *space, where* $n = |t|$

SST running time



Parallel Cost



Operations on DST

- All bioinformatics operations from Gusfield 1997 (for example) can be run on a DST without inter-node communication.
- The worse case time complexity is very poor.
- However, the average case time complexity is optimal in every case!
- In the paper we look at the LCS and exact local matching, maximal repeat finding, all-pairs suffix-prefix and exact set matching problems.

Summary of results for operations

Table 1: Post-construction average time complexities using standard and DST with k computing nodes. r is the number of strings.

Problem	Expected Running Time	
	ST (Serial)	DST (Parallel)
LCS and ELM	$O(n)$	$O(n/k)$
Maximal Repeat Finding	$O(n)$	$O(n/k)$
All Pairs Suffix-Prefix	$O(n + r^2)$	$O((n + r^2)/k)$
Exact Set Matching	$O(r \log n)$	$O((r \log n)/k)$

Load balancing

- In order to handle systematically biased data we need a method of load balancing.
- Choose a prefix length that is longer than the one that is needed and allocate more than one to each processor.
- Using this scheme we were are able to improve the efficiency from 61 to 89 percent while analysing human chromosome X on 16 processors. A prefix length of 3 was chosen and the 64 different prefixes were allocated using a simple greedy heuristic.
- Efficiency measured in terms of the size of the different SSTs.

Open questions

- Is it possible to construct a DST using $O(n/k)$ time and space per processor (where k is the number of processors).
- Can an SST for an arbitrary selection of suffixes be computed in linear time and in space proportional to the size of the SST?
- What is the running time of algorithms on a suffix tree that would require inter-node communication on a DST?