Project 2: Genome Scale algorithms

Maria, Mateo and Jacob

2/20/2018

Instruction of the project: https://github.com/mailund/gsa-exercises/tree/master/Project02

Project 2 - Suffix tree construction

Program description

The programs are implemented in Python (3) and we had no unsolved issues implementing the algorithms. Execution of the program on command line takes the form:

```
python suffix_tree_naive.py [text-filename] [string-pattern]
```

The output is directly to the console as a list of indexes of each first character of the pattern present in the text. If the pattern does not exist in the text, `None` is returned.

Test of search-suffix-tree-naive

We used the data provided for testing which outputs the following results:

python suffix_tree_naive.py mississippi.txt ss [3, 6]

python suffix_tree_naive.py banana.txt ana [2, 4]

We have also compared those results to the Naive exact match approach and they give the same indexes.

Time complexity

We have decided to evaluate the algorithm based on 3 different scenarios:

1. Tx-Pc: where the text is variable and the pattern is constant
2. Tc-Px: where the text is constant and the pattern is variable
3. Tx-Px: both text and patterns are variable

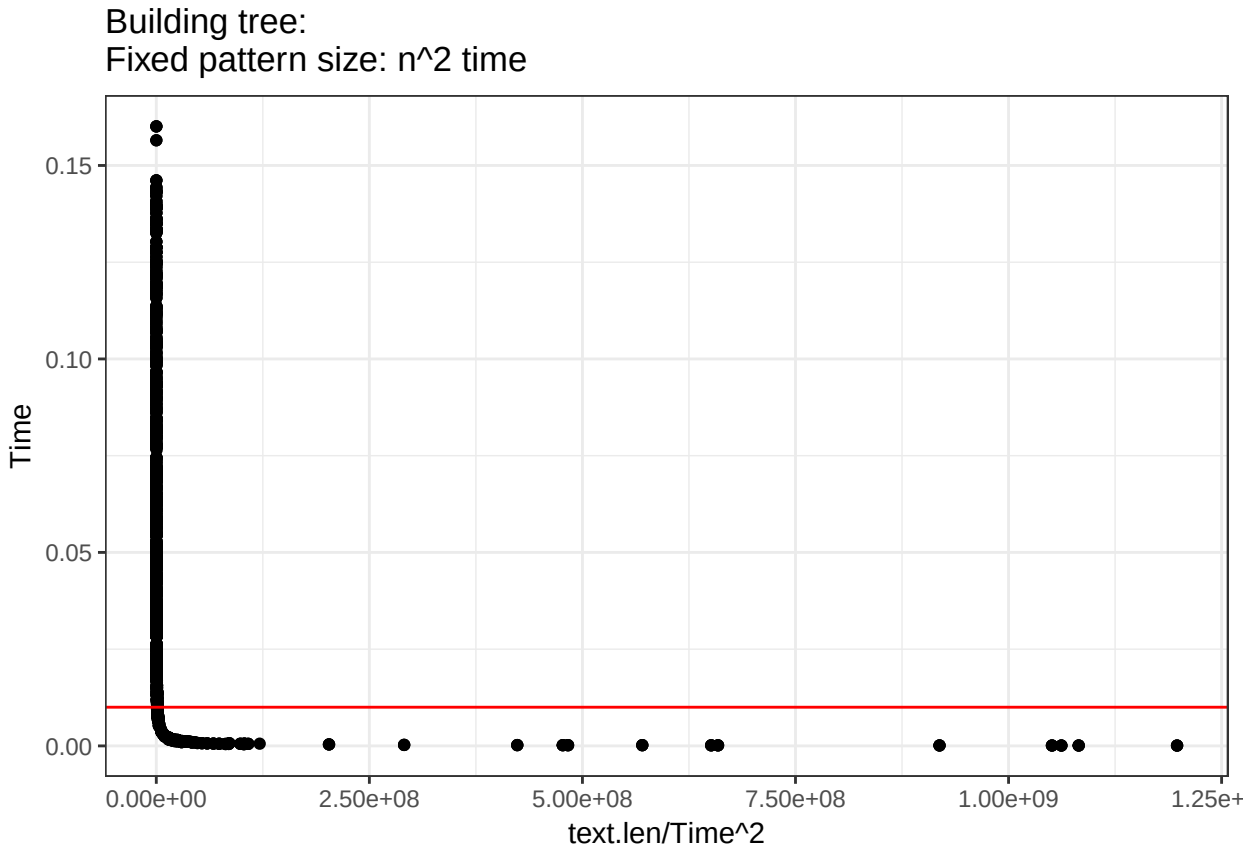
Using 3 different strings types:

1. aaa, alphabet size equal to 1
2. aab, alphabet size equal to 2, but just one character different in the end
3. abc, alphabet size equal to 62 (including numbers and Upper case letters)

Building the suffix tree

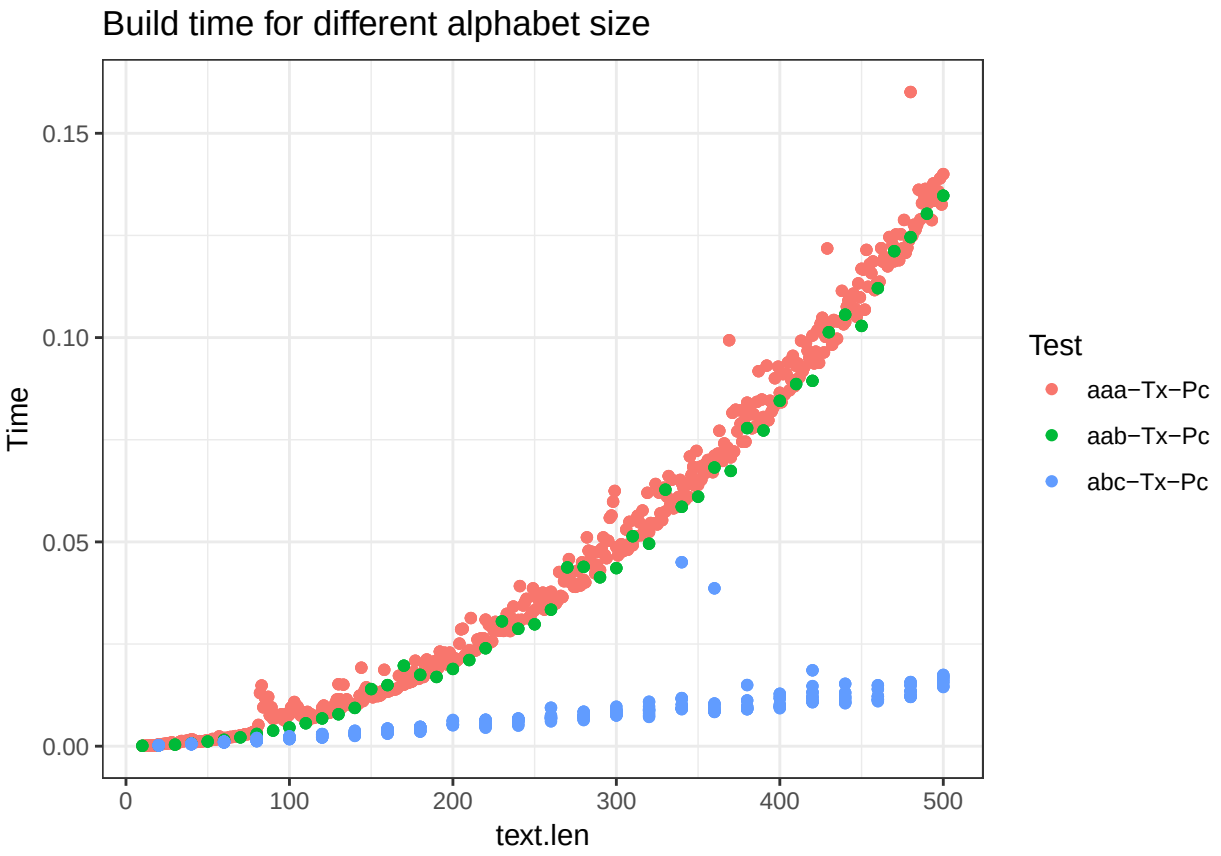
The implementation of the algorithm was made using the naive approach, which has $O(n^2)$ time construction complexity. To construct the suffix tree, each suffix is added at time, simply by comparing one character at time starting by the root of the tree. The worst scenario to build a suffix tree was made by creating a text T that has alphabet size $|\sum| = 1$. For example: 'aaaaaaaa$', where the sentinel character (`$`) is included in order to ensure that every suffix of the text `T$` is represented by a leave.

We increased the length of the text and stored the time of each tree-building operation. The best-case time bounds was construct by using text of `random` alphabet size, so one would not need to go through all nodes of the tree before it finds a tip to pull out the suffix.



In this plot it is clear to observe that the time complexity to construct the tree is quadratic ($O(n^2)$). This is because we need time $O(n-i+1)$ for the $i-th$ suffix and hence the total running time is:

$$\sum_{1}^{n} O(n-i+1) = \sum_{1}^{n} O(i) = O(n^2)$$



This plot shows that the alphabet size matters when constructing the tree. When the text is repetitive, one need to match through all the characters of the tree before pulling out the sentinel character (\$):

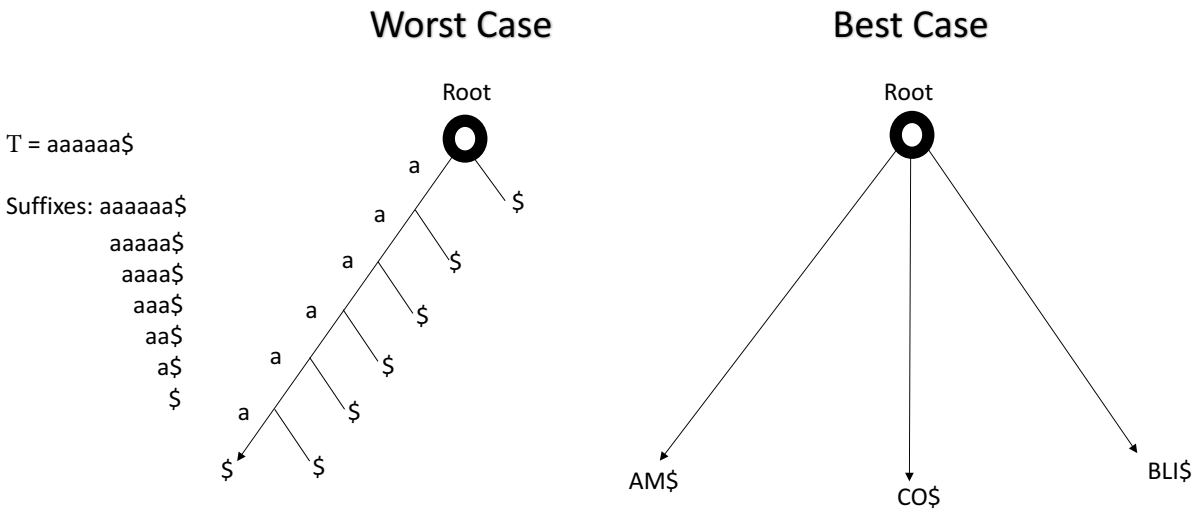


Figure 1: Best and worst case for the construction time.

Searching through the tree

Once the suffix tree for T is build, then searching through the tree for an exact match with a pattern P is trivial. Since all the substring of the text T is present in the suffix tree, it is easy to verify if a pattern P

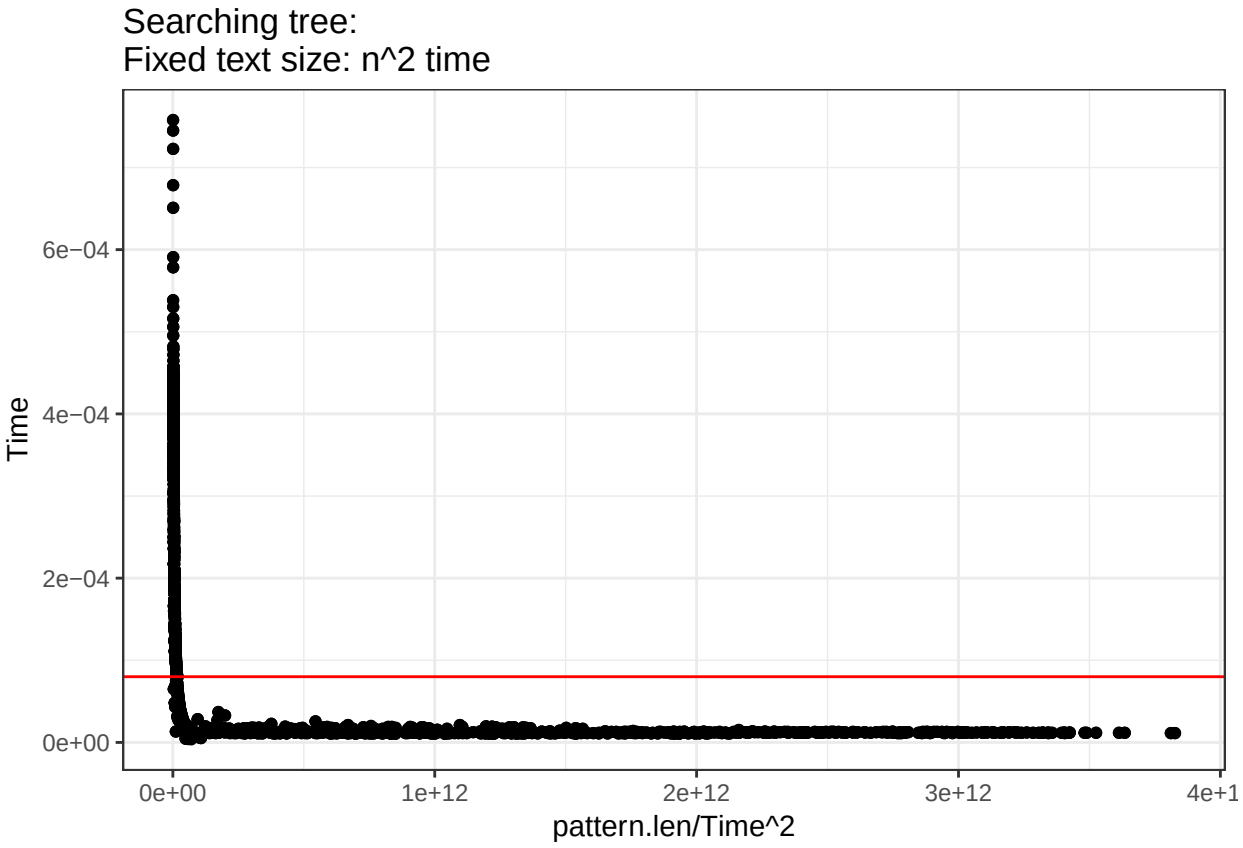
would be present in the text. It goes from the first character of the pattern $P_{[1]}$ until its last $P_{[m]}$, if one cannot proceed through the next character of pattern through the tree, it means that the pattern does not occur in the suffix tree/text. The operation of searching for the pattern through the tree takes $O(m)$ time.

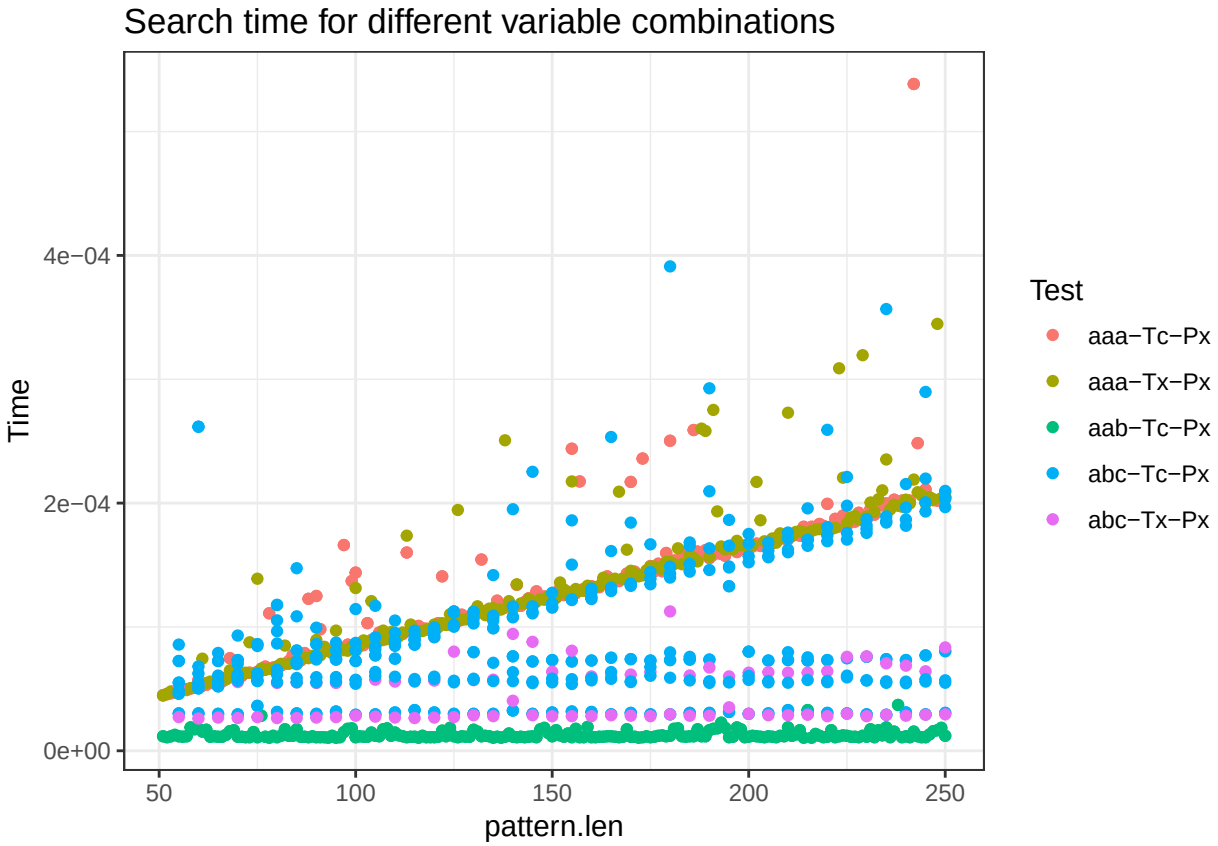
In the worst-case scenario the labels (suffixes) of the following path contain one character each, resulting in m child looks-up, which gives a time complexity of $O(m$ * time it takes to find the child edge). Let's define those look ups as j.

Therefore, the time for the search can be split in 3 different operations:

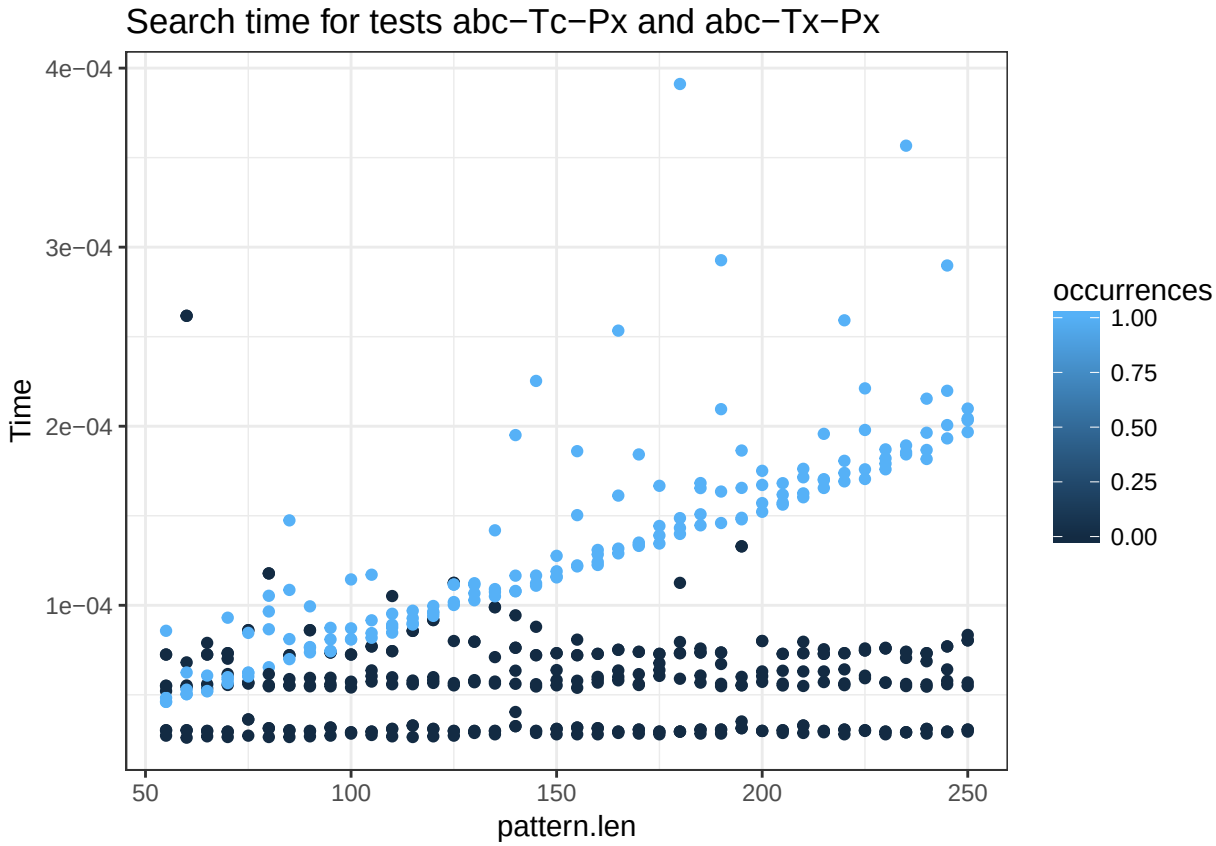
1. Finding the node where the pattern matches: $O(m)$
2. Collecting the leaves: $O(j)$
3. Both operation yield in: $O(m + j)$

As seen in the following graph:





Not taking into account the number of occurences of the pattern in the text, it is possible to see that the time to search is proportional to the depht of the search. If the text is repetitive, one would need to go deeper in the nodes. If the text is random, it is unlikely that one would need to go that deep before finds a match or before a mismatch.



You can see through this graph that time grows with the existence of the pattern in the tree.