Compact suffix trees

Abstract

In this report, a method will be showed to construct compact suffix trees using Ukkonen's algorithm. It will be shown that the building of such a tree has a linear runtime complexity with respect to the string length, using empirical experiment results. Furthermore, a technique will be shown that utilises suffix trees to search strings. It will be proved using timing experiments that such a technique has a linear time complexity with respect to the search term length as opposed to the length of the string to be searched, which is the more frequent case with other search methods. Furthermore, it will be shown that all matches of a substring can be found also in linear time.

Introduction to suffix trees

Suffix trees are a type of search tries that allow for the implementation of particularly fast, efficient algorithms related to strings such as search, regex matching, finding the longest repeated substring, etc. Since gene and protein sequences can be represented as large strings, which need to be analysed in many ways, bioinformatics uses this data structure frequently. What a suffix tree tries to accomplish is to make all the possible suffixes of a string searchable. For example, the string, "BANANAS" has seven suffixes:

1. "BANANAS"
2. "ANANAS"
3. "NANAS"
4. "ANAS"
5. "NAS"
6. "AS"
7. "S"

It is easy to see from the above example that a naïve implementation of a string-searching algorithm that stores these suffixes to search the string would have a quadratic space complexity. Furthermore, it would have a quadratic time complexity as well, mainly because partial matches (for example when searching for "NAS", there is a partial match at suffix 3, until the character A). Suffix trees avoid this by compacting matching prefixes of suffixes into one edge (e.g. "ANAS" and "AS" start with "A" this will be only one edge). A side effect of this behaviour is that there is a one-to-one mapping between the leaves of the tree and the ending of the encoded string. This is a very desirable property as this makes it possible to construct efficient solutions to many problems encountered, when processing strings.

An example of a suffix tree encoding the suffixes of the string "BANANAS" is shown below.

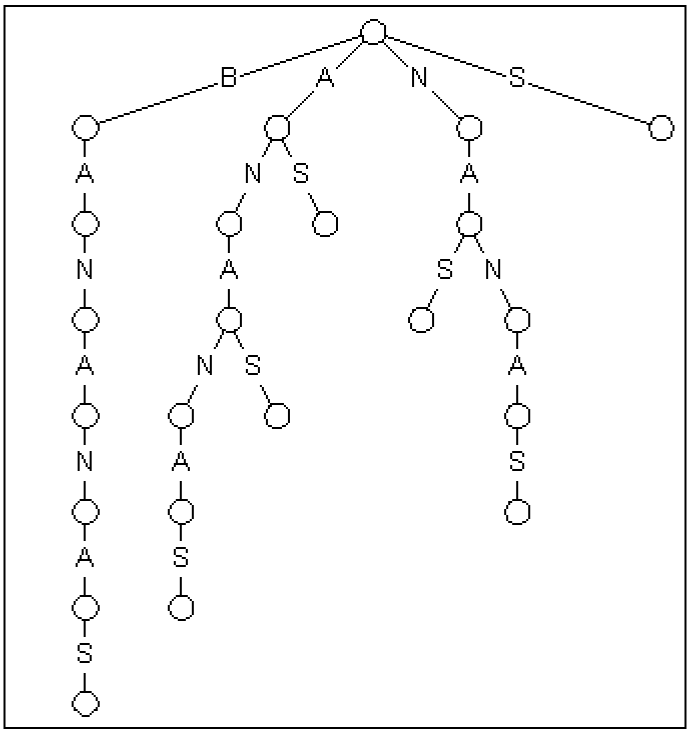
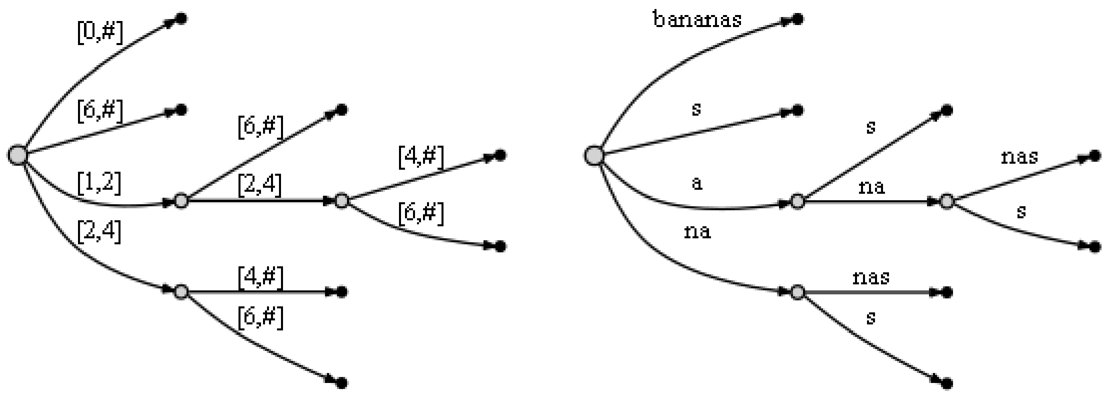


Figure 1Basic suffix tree of "BANANAS" [1]

It is easy to see from the example above that a suffix tree's most important property is that the edges leaving form each node represent the only possible suffixes from that node. For example following the edges N-A it is clear that this can only be followed by two suffixes "S" and "NAS". In addition, if one traverses the whole tree up until the leaves, all the suffixes of the original string can be reconstructed. From these properties, it is easy to construct a string-searching algorithm. The search starts from the root node. For each character of the search, we move to the child node matching the next character of the search term. If there is no child node matching the next character, then the term is not a substring of the text. If the search does not terminate at a leaf node then we traverse through the leaves from there, and each leaf will correspond to one match. There is an issue with this solution: since the edges were labelled with the literal characters, we do not know where these matches are in the original string.

This is where compact suffix trees come in. There are only a few differences between a compact and regular suffix tree. Firstly, edges are not labelled with the literal characters of the original string, but with starting and ending position pointers in it instead. Secondly, nodes with a degree of one (i.e. nodes with only one child) are compressed into a single node. All other properties are retained in a compact suffix tree. Therefore, the path B-A-N-A-N-A-S will be compressed into a single node containing [0, #] (# means until end of string). It is easy to see that this is much more space efficient. The compacted suffix tree for the word "BANANAS" is shown below (with pointers on the left and pointers resolved to strings on the right).



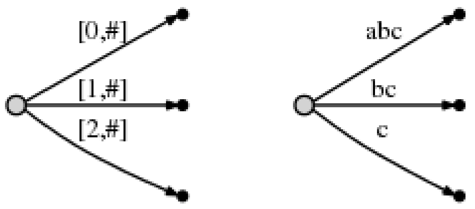
In the rest of this report, the term “suffix tree” will refer to the compact suffix tree structure. This is because the non-compact suffix tree (also known as suffix trie) structure is rarely used since it can only be constructed in O(n^2) time.

CONSTRUCTING SUFFIX TREES

The following few paragraphs will try to explain how suffix trees are built in a very informal fashion. The algorithm to construct the tree is based on Ukkonen’s method [2] first developed in 1996. This algorithm was the first to construct suffix trees in a left to right (aka. online) fashion. Ukkonen developed this algorithm based on McCreight’s work on suffix trees [3], 20 years earlier, in 1976. Furthermore, Ukkonen’s algorithm is a lot easier to understand, that is why this algorithm was chosen to construct suffix trees. Since then many research studies were focused on the data structure, there are newer, lot more efficient algorithms (e.g. ERA [4], B^{2}ST [5]) constructing this data structure, but Ukkonen’s is still the most popular algorithm chosen for string processing. For a more rigorous explanation including proofs of the algorithm’s correctness and runtime complexity, see chapter 6.1 of Gusfield’s book on string algorithms [6].

The algorithm takes one character at a time from the string that has to be processed and adds those characters one by one to the tree. At the end of each step, the tree will contain all the suffixes of the string until that position. To investigate the algorithm further let us look at the example of generating the suffix tree of the string “abcabxabcd”.

The first three steps are quite easy: a node is inserted for each suffix with different prefixes (a, b and c). The tree at this point is shown below, with pointers on the left (# representing the “until current position” pointer) and resolved on the right.

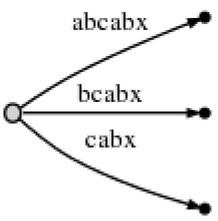


When b and c is inserted there is no work done to update the previous nodes, we get this for “free”, since these nodes end at the current position, therefore they are updated automatically in constant time. This is called implicit addition. To go further some variables have to be introduced:

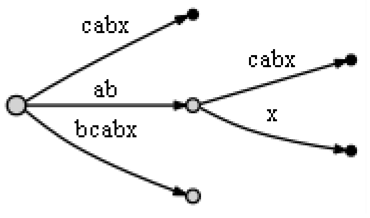
- activeNode – stores the currently processed node, this is where new nodes will be inserted
- activeEdge – stores the currently processed character; this is not always the same as the new character that needs to be added to the tree, because of implicit additions
- activeLength – stores how many implicit additions have to be processed
- remainder – stores how many suffixes need to be added to the tree, this can be more than one, again because of implicit additions

In the next step, when ‘a’ is inserted the algorithm does not add a new node since there is already an edge starting with ‘a’; activeEdge is set to ‘a’ and the activeLength and remainder are incremented. After this step, the tree is unchanged in memory since this node is implicitly contained in the tree. The same happens when ‘b’ is inserted. This time, however, we have to process the string “ab” as this is left over from the last step. Though, again, the suffix ‘b’ is already implicitly contained in the tree, therefore it is not updated only the variables are changed: activeEdge to ‘b’, activeLength and remainder are incremented.

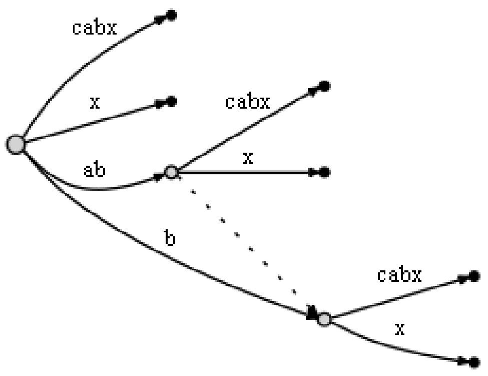
Following this, ‘x’ is inserted. When this happens, the remainders of the previous steps have to be processed since this is a new suffix. After the implicit update, the tree looks like this.

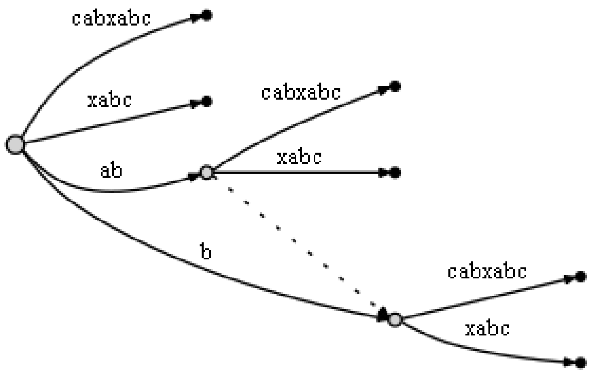


This tree is incorrect though, since the string “ab” can be followed by “x” and “cabx”. For this, a new operation is applied: split. The active* variables tell the algorithm where exactly to split a node and the tree is updated to the following state. The remainder is decremented.

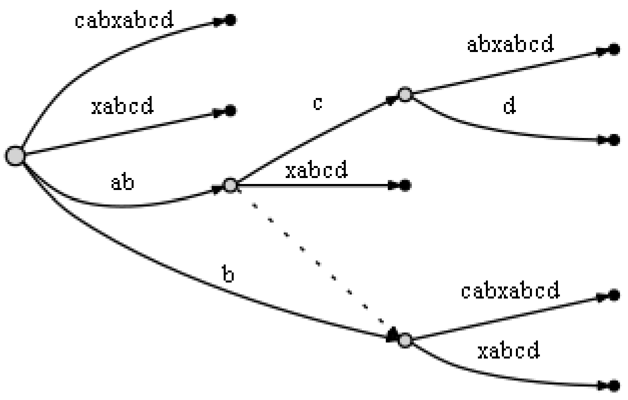


After this, the remainder is still two; this tells the algorithm to insert two more suffixes: “bx” and “x”. This is done in a similar fashion as above, however when the node labelled with “bcabx” is split; a suffix link is inserted pointing towards it, coming from “ab” (shown with a dotted arrow). A suffix link in essence, shows the algorithm where similar parts of the tree are located and makes it possible to keep the updates of the tree very efficient. Gusfield’s book, referenced above, provides rules explaining why, where and when these suffix links are needed. As a last step the suffix “x” is inserted; this only yields a new node from the root. The state of the tree at this point is shown below, on the left.

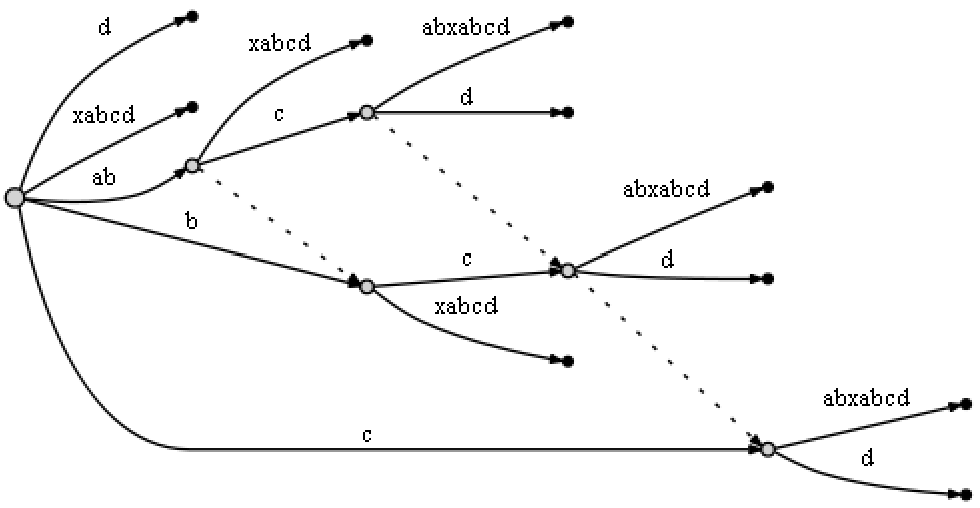




The next three characters (“abc”) are inserted implicitly there are no special cases here. The state of the tree is shown above on the right, for clarity. An interesting thing happens, though, when ‘d’ is inserted after this. Firstly, the node labelled with “cabxabc” is split; this is no surprise though, if we follow the above-mentioned rules. The surprise is that after this the activeNode has to be updated to where the suffix link points. The state of the tree after this is shown below.



After this the remainder is decremented to three, therefore “bcd”, “cd”, and “d” still have to be inserted. “bcd” causes a split at the current activeNode, and inserts a new suffix link. “cd” causes a split at “cabxabcd” and receives a suffix link as well. Finally, the solitary “d” node is inserted. The final state of the tree is shown below.



To conclude the explanation we can say that this algorithm has a runtime complexity of O(n) where n is the length of the string that has to be processed. This is because a single step has to be taken for each character of the string and each character can be processed in constant time. The reason for this is that “algorithmic tricks”, such as suffix links make it possible to avoid the re-traversal of the tree.

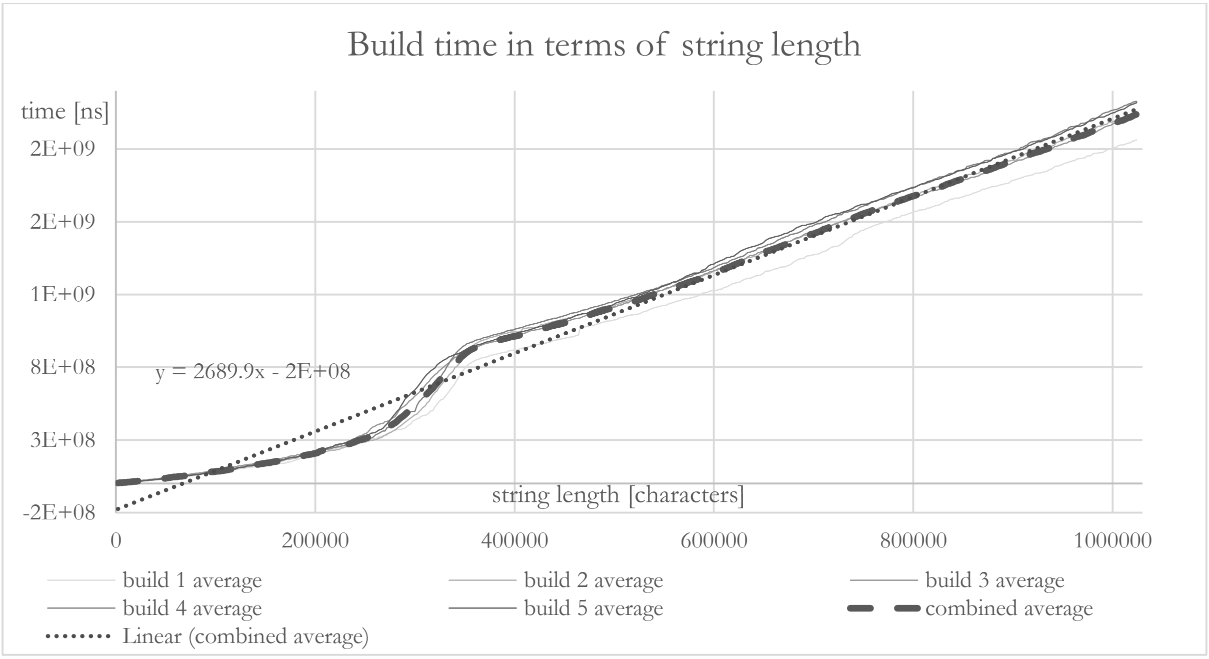
Analysis of the runtime complexity

To prove that the algorithm runs in O(n) time an experiment was constructed. The algorithm was implemented in Java. This decision was made firstly because the language’s object oriented nature makes it easy to construct recursive trees. Secondly, it is easier to make timing possible in imperative/object oriented languages rather than in functional ones. One drawback of it is that the non-deterministic nature of the Java Virtual Machine (garbage collector) makes it harder to make accurate measurements of the runtime of the program. On the other hand, this would be the case as well in a deterministic language such as C since the running of the program would occur in a multicore, multithreaded environment where system interrupts and random context switches can slow down the operation of the program. To counter the non-determinism of the garbage collector, collection was forced in the Virtual Machine in some experiments. Comparing the data when garbage collection was forced as opposed to ones where it was not, this method provided results that were more accurate.

Since suffix trees are frequently used in bioinformatics, it seemed appropriate to use a genome sequence (first 1000 base pairs of the rat genome from the Santa Cruz University bioinformatics database [7-9]; ~14M characters) for these experiments, as test data. To confirm that the algorithm works with the same efficiency on natural language strings the full text of Leo Tolstoy's War and Peace (~3M characters) was used as well.

The experiment runs as follows. A text file (one of the above-mentioned sources) is loaded into memory as a string. Then the string is split apart into a predefined number of pieces. The splitting was applied to avoid anomalies in the runtime of the algorithm caused by the structure of the string. Then a suffix tree was built for each of these parts. Measurements were taken after every nth (predefined) character was processed in the string. The tree for the same substring was built several times, and then the timings for each run were averaged. This method was applied to smooth out anomalies caused by the multithreaded environment. Before and after each time a tree was built a garbage collection was forced, the reason for this was explained above. The timings were taken using the method `System.nanoTime()` to make it as accurate as possible.

After the whole process was complete, the timings were written to a CSV encoded text file and were post-processed using Excel. The result of this experiment is shown below.



The thin lines show the average of building each split twenty times, the thick, dashed line shows the average of all of the runs. The straight line is a linear trend line of the combined average. It is easy to see from this graph that indeed the algorithm runs in linear time. It is not clear why the slow-down slope occurs every time at around 300000 characters. A reasonable explanation would be that this is caused by excess memory allocation by the VM. Since garbage collection is forced before a build starts, the VM also deallocates some of the memory that was allocated for the previous run, and the tree and collected timing data only gets big enough at around 300000 characters to force the VM to allocate extra memory. This would also explain why the first build always runs slower than the following ones do. Even with this hard-to-explain temporary slow down the runtime of the program still shows a clear linear correlation. If the line following the slowdown were extended towards the origin, the line would cross the x-axis at the origin, unlike the trend line. This further proves the strong linear correlation.

Analysis of the runtime of searching

Since suffix trees are most commonly used to make searching of strings very efficient it seemed appropriate, to not only prove its build efficiency, but also the searching efficiency. Another experiment was constructed to show this. This experiment tries to prove two things (assuming that the suffix tree was built for T of length n in O(n) time):

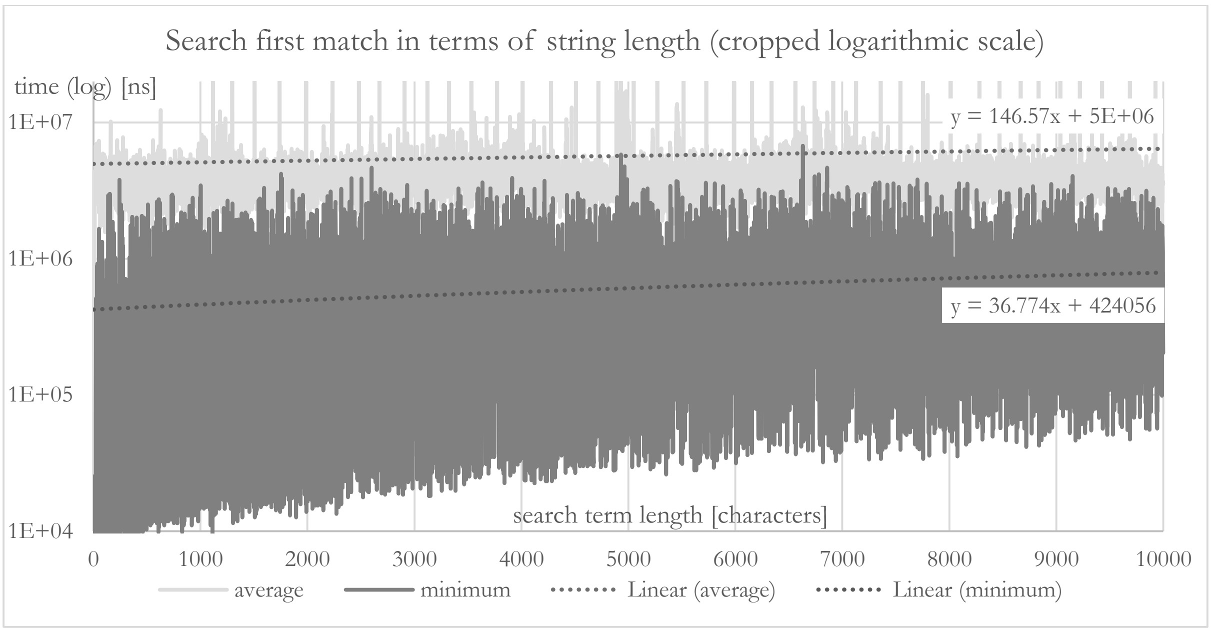
1. that the first match of pattern P of length m, which is a substring of T, can be found in O(m) time,
2. and that, all z matches of P can be found in O(m+z) time.

To prove this the following search algorithm was constructed. After the tree is built, the following method is used for searching. An empty TreeSet is constructed; this is where the results are going to be collected. Then the following happens recursively. The character at the current position in the search term is taken, and is compared with the outgoing edges from the current node (starts with the root node). If an edge matching this character is found then the rest of the edge's label is compared with the term starting at position. If these match then either a recursive call happens because the search term is not yet exhausted (position not at end of term), or the leaves of the tree from that point are searched. This is done using a simple depth first search. Using the collection of leaves the starting positions can be determined of the matched search term. The algorithm can be set up so that it returns after finding the first leaf.

Using this search technique the experiment was constructed. First, one of the data files is loaded into memory as a string. Then using this string, a suffix tree is built using Ukkonen's algorithm, constructed earlier. This tree is then used for repeated string searches. These repeated string searches increase in length until a predefined point. For each string length a predefined number of searches happen, each with a random substring of the current length. Timings are taken again with the `System.nanoTime()` method to be as accurate as possible. The time it takes to find the first result, and the time it takes to find the last result (with the number of results) is recorded.

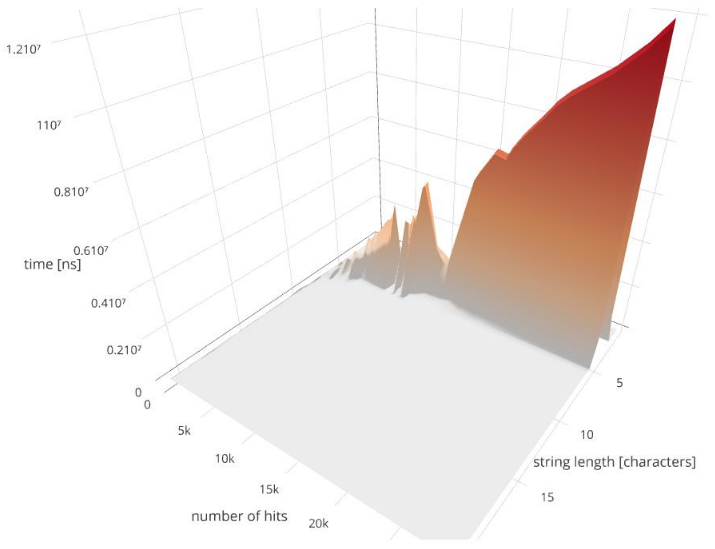
After all searches are done the timings are written to two separate CSV files: one for the timings of the first result (called firsts), and one for the last result (called lasts). Whether the data is collected for each of these files can be easily turned off. The file, firsts is fairly straightforward; a row for each string length and a column for each run of searches. The second file, lasts however, is a bit more complicated. A row corresponds with the number of results (the first column is labelled with the number), and each column corresponds to the length of the string. Timings for only one result are not recorded in lasts, this would be unnecessary, since this data can be found in firsts. The table works like a coordinate system, therefore, a cell with the row label 15 and column label 10 that has the value of 13422 means that finding all 15 positions of a string that was 10 characters long took 13422 nanoseconds. The reason why this output structure was chosen was that this seemed to be the only way could it be proved that finding all results has a O(m+z) complexity, only later was it realised that this complicated structure was not necessary.

The graph below shows the experimental results for finding the first matches. The graph is in a cropped logarithmic scale on the y-axis to better illustrate the growth of the runtime complexity.

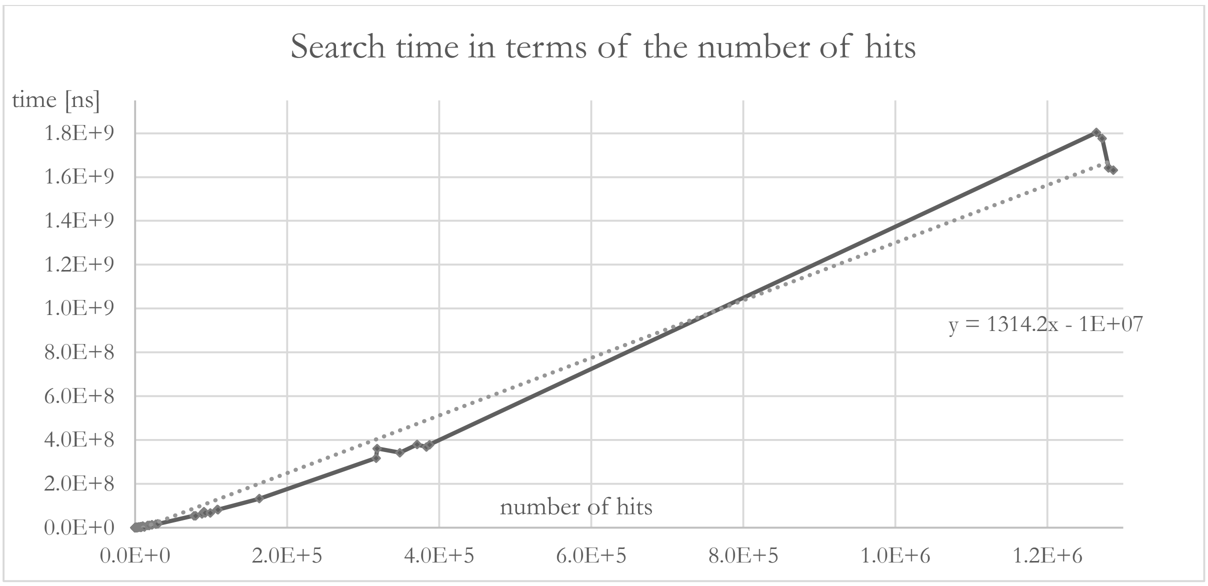


The graph is based on the minimum and average of 10 searches of increasing length up to 10000 characters. In regular, linear scale, it is not immediately apparent that the average time grows, because of the great variance between runtimes. However, switching over to logarithmic scale, with some trend lines immediately shows how the time indeed grows linearly with respect to the string size. Notice that the equations for both trend lines have a really low coefficient constant. What this translates to in practice is that finding the first match is near instantaneous; the data shows that even in the worst case it only goes up to a single second range but even this case is probably only caused by garbage collection.

The next graph shows how the number of results and string length affect the search time.



The graph can be misinterpreted as the string length having a reverse correlation with respect to the search time. This is not the case, however. The truth is simply that longer strings yield fewer results, which take less time to find. The next graph makes this clearer.



From this graph, it is obvious that there is a linear correlation between the number of hits and the time it takes to find them. Notice the equation for the trend line. This time it has a coefficient of 1314. Compare this with the results on finding the first hit in the tree; there the coefficient was only in the 100 range. From this it can be said that finding all results has a runtime complexity of O(m+z), since finding the first match takes O(m) and only then can the other leaves be found adding up to O(m+z). However, it is clear from the experiment that z (the number of hits) dominates the expression, and it can be argued that m should be omitted from the expression since z is overly dominant.

CONCLUSION

In this report, it has been shown how Ukkonen's algorithm works, that construct compact suffix trees in an online fashion. It has been proved using empirical experiments that such an algorithm can construct a suffix tree of text T of length n in O(n) time. Furthermore, a search technique was demonstrated that uses suffix trees to search in strings. It has also been proved that the first match of a substring of length m can be found with this method in O(m) time; again with timing experiments. Lastly, it has been shown that all z matches of substrings can be found in O(m+z) time.

REFERENCES

[1] M. Nelson, "Fast string searching with suffix trees," *Dr. Dobb's Journal*, F. 1, ed., 1996, pp. 115-119.

[2] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica,* vol. 14, no. 3, pp. 249-260, 1995.

[3] E. M. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM (JACM),* vol. 23, no. 2, pp. 262-272, 1976.

[4] E. Mansour *et al.*, "ERA: efficient serial and parallel suffix tree construction for very long strings," *Proceedings of the VLDB Endowment,* vol. 5, no. 1, pp. 49-60, 2011.

[5] M. Barsky *et al.*, "Suffix trees for very large genomic sequences." pp. 1417-1420.

[6] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*: Cambridge university press, 1997.

[7] R. A. Gibbs *et al.*, "Genome sequence of the Brown Norway rat yields insights into mammalian evolution," *Nature,* vol. 428, no. 6982, pp. 493-521, 2004.

[8] B.-H. A. group. "Jul. 2014 assembly of the rat genome," http://hgdownload.cse.ucsc.edu/goldenPath/rn6/bigZips/.

[9] P. Havlak *et al.*, "The Atlas genome assembly system," *Genome research,* vol. 14, no. 4, pp. 721-732, 2004.