Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Counter based suffix tree for DNA pattern repeats

Tshepo Kitso Gobonamang, Dimane Mpoeleng

Department of Computer Science and Information System, Botswana International University of Science and Technology, Botswana

A R T I C L E   I N F O

Article history:
Received 26 December 2018
Received in revised form 28 November 2019
Accepted 16 December 2019
Available online 15 January 2020
Communicated by T. Calamoneri

Keywords:
Suffix tree
Bioinformatics
Counter based suffix tree
Nucleic bases
Genes
Genome

A B S T R A C T

In recent years, the string datasets have increased exponentially, so is the need to process them. Most of these datasets have been deeply rooted in the field of bioinformatics since the entire characteristics of any living organism is encoded in their genes. Genes consist of nucleic bases which will, therefore, makeup the entire genome. A genome is made of a concatenation of different types of nucleic bases. To efficiently extract the information encrypted in these sequences there is a need to use algorithms to decrypt it. Most available methods use the data structure commonly referred to as the suffix tree. They have tremendously evolved over the years, and the on-line construction of the suffix tree is deemed as the best data structure, however, it is not optimal when it comes to finding repeated sequences because of many traversals algorithm will have to do when identifying repeats. To improve the speed and of finding repeats we developed a counter based suffix tree algorithm. Our work presents a novel algorithm of constructing a counter based suffix tree without losing its properties. The counter based suffix tree time complexity is $\theta(n)$ where n represents the length of a string. Which is the same as the fastest suffix tree implementation. We have shown that the counter based suffix tree will reduce the search time when identifying repeats. We have proved that a counter based suffix tree can be developed during construction.

1. Introduction

A human DNA molecule which denotes both the phenotype and genotype of human can be simplified as the repeated, concatenation and a mixture of characters ($A\ C\ T\ G$). They each represents a certain compound, A represents Adenine, C stands for Cytosine, T denotes thymine and G represents Guanine. The human DNA is a combination and repetition of these bases and it has the length of around 3.5 billion characters. Therefore, analysing such a long string demands a well thought of data structure in order to find patterns accurately in a short period of time. There is a clear implication that according to computational biologists a good data structure is the one which can find solutions for the following queries;

- Given a certain combination of compounds in a genome find if there exists a certain sub combination within the main one. Since we know that the compounds in question can be expressed as strings then we can simplify the query to say “Given a string S, find if string T is a sub-string in S”. For example during the process of comparing Neanderthal genome with the modern human one there would be a process where the human genome sub sequence was compared against the sub strings in Neanderthal genome. This was a complicated process since there will be a lot of computations that is why the parallel processing was needed [1].

E-mail addresses: tshepo.gobonamang@studentmail.biust.ac.bw (T.K. Gobonamang), mpoelengd@biust.ac.bw (D. Mpoeleng).

https://doi.org/10.1016/j.tcs.2019.12.014

- Given string S find if T is a suffix of S. During the enzymes restriction maps of a human akt loci in AKT1 and AKT2 finding suffixes of each sequence was a principal determining factor [2].
- Given string S find if sub-string T is repeated else where in S. Alec J Jeffreys and fellows concluded that a human genome has many *minisatellites* that are highly harmonic due to the fact that they have allelic variations in repeat copies for each *minisatellite*. To find these, this query had to be answered throughout this process [3].
- Given a string S find the longest repeated sub string. Recognising irregularities in repeat length is necessary since this might help in discovering some disorders that have been proved to cause Fragile X syndrome, myotonic dystrophy and friedreichs ataxia [4]. This process will involve finding the longest sequence in the given human genome.

2. Repeat identification and significance

This section reviews search strategies imposed on suffix trees to find repeated sequences and how those strategies are performed on already created suffix trees. A significant number of efforts have been directed in this discipline because the presence of repeats in human genome has recently captured the interest of most bioinformatics.

Zheng and Lonardi [5] gave a definition of elementary repeats that incorporates both the frequency and length of repeated sequences. They defined the elementary repeats of sequence T as a substring of S, given the length l and frequency f, substring S which is non-trivial, will have a maximal length greater than l occurring at least f times. The frequency of every substring A whose length is greater than l should be the same as the frequency of S, therefore A is a sub-repeats of S, elementary repeats are actually the basic building blocks of these sequences. If there are a lot of elementary repeats in long repeats, then there will be a lot of complex structures of repetitive patterns.

Zheng and Lonardi [5] also claimed that the discovery of these internal components can be helpful in inferring the roles of repeats. The conservation of elementary repeats could reveal the evolutionary relationships amongst different types of repeats. Therefore it is essential to find elementary repeats, and if these repeats are the same they will be referred to as exact elementary repeats and if there is an allowance for differences then they will be referred to as appropriate elementary repeats.

Zheng and Lonardi's algorithm [5] uses the bottom up strategy. To build a suffix tree they used a linear time algorithm proposed by Dan Gusfield [6] which is the same as the Ukkonen algorithm explained earlier. After the construction of the suffix tree they fixed the frequencies of all $n-l+1$ seeds whose length are l (called as l-mers seeds). These seeds start at positions $T[i]$ to $T[n-l+1]$ respectively, thus it merges the successive seeds with equal frequencies unto intervals. Afterwards it extracts all the intervals consisting of seeds the equal frequencies greater than f, and then for each interval they check the corresponding occurrences of every single pair of successive seeds using the suffix tree.

They find successive seeds using the following theorem:

> A non-trivial substring A which occurs at least twice, is an exact elementary repeat if and only if it is a maximal non-trivial substring such that all its l-mers are as frequent as A itself and if their intervals is also successive in all other copies otherwise this interval is divided into two pieces and then checked again. Checking every pair of occurrence takes $\theta(n)$ time. The comparison of all relative positions for every pair of two successive seeds takes about $\theta(f)$, since each pair has to be checked at most once since there are $\theta(n)$ pairs of such pairs of seeds, the time complexity of this algorithm tends to be $theta(n^2 f)$. This is due to the fact that the algorithm needs to check all $n-1+f$ seeds from a suffix tree.

As evidential from the explanation above this algorithm involves two processes which is the construction of the suffix tree and the searching processes which is done on the created tree by the Dan Gusfield algorithm [6]. The construction of the suffix tree does not in any way involve any modification to the original Ukkonen algorithm.

Most algorithms produced on the study of repeated sequences depended mostly on the use of annotated library of repeats. In the library of repeats, known repeats are manually collected and later on, they can be accessed. Smit and Green [7] in 1994 developed an algorithm known as the RepeatMasker which uses the annotate library. The RepeatMasker steps through every element in the string database and match the sequences with what is in the library. The same procedure goes for BpMatch [8] and RepScout [9] the only difference is the speed of recognition and the method of extracting the repetitive patterns in the string database. The BpMatch is an algorithm that works on a suitably modified suffix tree structure and its able to compute the coverage of the source sequence T on the target sequence S, by taking into account direct and reverse segments eventually overlapping.

BpMatch [8] actually outputs the important segments found and the computed segment-based distance. The RepeatScout [9] algorithm uses the *De-novo* repeat family identification and focuses on multiple alignment strategies. The main issue with RepeatScout is to identify repeat families present in large sequenced genome. The RepeatScout identifies the family of repeats via extensions of the consensus seeds. The method enables a clear definition of repeat boundaries which is important in repeat analysis. The limitation in all of this method is that these algorithms cannot treat new genomes. With the exception of the RepeatMasker the other methods cannot give a good representation of the length, the frequency and the location of repetitive patterns.

The other algorithms are the modification of the earlier discussed algorithm since they don't use the library of repeats and the can actually treat new genomes. With these algorithms the new repetitive patterns can be discovered and there is no need of manually updating the library. The only difference amongst these algorithms is the biological representation of data for example the REPuter [10] which was carried out by Kurtz and Schleiermacher defines repeated strings as a pair of similar strings of maximal length. The tool computes exact repeats and palindromes in the entire genome efficiently. The REPuter consists of two programs which are a search engine which actually processes a DNA sequence given by the user in a Fasta-format and returns a representation of all maximal repeats, and the visualizing component which processes the output of the search engine and provides an overview of the number, the length and the location of the repeated substring. However the algorithm represents only the pairs of repeats and hence does not consider the frequency of repeats and locus of repeats.

The Suffix ExactRepeat [11] is an improvement to the REPuter [10] as it gives a representation of length and the frequency of the repeated sequences. Processing is the same because it uses a suffix tree based database strings but Dan He's algorithm uses the elementary repeats proposed by Zheng and Leonardi [5]. In this algorithm the first thing to find out is the elementary repeats and refer to those repeats as the ExactElementary repeats and will later on be used. This algorithm extracts exactly one copy of possible candidates of elementary repeats and then tries to prune the false positive candidates effectively. The algorithm can be extended to find more complex repetitive patterns. The algorithm proved to be able to report all repetitive patterns that are nontrivial and what elementary repeats they contain. Although the algorithm is more efficient algorithm in spatiotemporal terms, it leaves out the most important factors such as the location of repetitive patterns and the algorithm focuses only in finding the elementary repeats and leaves out the appropriate elementary repeat.

RepSeeker algorithm by Hongwei and Xiaowu [12] tends to find all the elementary repeats of the input sequence T and then outputs the repeated sequences in a sorted manner characterizing the lists with the starting position and the end position of all strings to make sure that the position or location of each repeated string is known. The RepSeeker algorithm makes use of the Ukkonen suffix [13] tree construction algorithm. The RepSeeker algorithm does not allow for querying the string database to check for repeated sequences.

The last algorithm to be discussed is the Bruijn graph [14] which finds the shortest circular superstring that contains all possible substrings of length k over a given alphabet. The algorithm although can locate the repeated strings takes time and it's not easy to analyze. Unlike a number of previously discussed algorithms the Bruijn does not use a suffix tree algorithm. The other limitations of this algorithm are that it does not state the frequency and the location of the repeated sequences.

3. Construction of a suffix tree

A number of efforts have surfaced in relation to creating a disk resistant suffix tree. One of these efforts entails the Hunt's algorithm [15] which was deemed to be the first disk resistant suffix tree for a human chromosome. In the efforts of achieving better locality of reference this method removed the use of suffix links. His algorithm also introduced the pre-fix based partition method for the construction of disk based suffix trees. The length of these prefixes are incremented untill the partition corresponding to each prefix fits into the main memory [16], hence all prefixes must have the same fixed lengths. However, this approach couldn't handle the skew present in data which results in suffix tree partitions having various sizes. Due to the removal of suffix links, the complexity of this method is increased to $\theta(m^2)$ from $\theta(m)$, where m is the length of the index string from which a suffix tree is to be developed. After this effort a lot more followed, here are just a few of those methods which significantly enhanced the process of a suffix tree construction.

A buffering strategy called the TOP-Q proposed by Bedathur [17], built the disk based suffix tree with suffix links for larger sequences. The TOP-Q is built upon the linear complexity of the Ukkonen's algorithm, and its main contribution was to show that a complete suffix tree can be built for larger sequences. In addition to that, TOP-Q has shown that an array representation of suffix tree nodes is more efficient than the linked list representation in terms of space efficiency.

Tata [18] came up with a similar approach to that of Bedathur, but they abandoned the use of suffix links to achieve better locality of reference, which resulted in their algorithm having the time complexity of $\theta(m^2)$, for construction. Amazingly their algorithm was the first to scale up to a human genome. This approach was able to significantly reduce the random disk accesses which resulted in its better performance than Hunt's algorithm and TOP-Q in constructing the suffix tree.

Although Tata's algorithm was able to scale up to a human genome theoretically, the first complete genome scale disk based suffix tree was constructed by Phoophakdee [19] and the algorithm was called the Trellis. To handle skew present in the genome sequence they used partition and merge approach with variable length. Initially Trellis algorithm constructed the suffix tree without suffix links but its advantage was that the suffix links could be recovered hence its time complexity was $\theta(m^2)$ to traverse through nodes during construction. Due to its operations being in-memory it avoided the random disk accesses. An upgrade of Trellis is the Trellis+ [20] which is much faster and it has some memory constraints, but it also removes the restrictions of keeping the whole indexed sequence in-memory.

Trellis gave other researchers an idea of constructing a genome scale suffix tree. Barsky proposed DiGeST [21] algorithm to construct the suffix tree, and this algorithm had the ability to handle sequences that are much longer than the human genome. The algorithm first partitions the indexed sequence and each partition is sorted lexicographically, thereafter the sorted partitions are merged with the help of suffix arrays to construct the final suffix tree. This algorithm also does not use suffix links, hence it would not minimise the search efforts if suffix links are discarded as it is the case with other algorithms discussed so far. An upgrade of DiGeST is B^2ST [22] which was also proposed by Barskey. Although it discarded the use of suffix links, it can construct a suffix tree for the text sequence of 12GB. Its main contribution is that there is no need to keep an indexed sequence in the main memory during the construction of the suffix tree.

Algorithm 1 High Level Counter Based suffix tree.

```
1: Construct tree $T_i$
2: for $i \leftarrow 1$ to $n-1$ do
3:     Begin [Phase $i+1$]
4:     for $j \leftarrow 1$ to $i+1$ do
5:         Begin [extension $j$]
6:         Find path from root labelled $S[j \cdots i]$ in the current tree, if needed extend the path by adding character $S[i+1]$, if string $S[j \cdots i+1]$ is in the tree. If a node is created adjust counters accordingly.
7:     end for
8: end for
```

A much speedier suffix tree construction came recently and it is called the WAVEFRONT [23] which was proposed by Ghoting. The WAVEFRONT construct the suffix tree by dividing it in disjoint tiles which are much similar to diagonal partitions of the suffix tree instead of vertical partitions. WAVEFRONT algorithm performs better than other suffix tree construction algorithms when the indexed sequence does not fully fit on the main memory. On high performance systems the WAVEFRONT will construct the suffix tree in a very few minutes.

3.1. Search optimized suffix trees

This section reviews the efforts made on search optimization of the suffix tree. It is very important to review these efforts because as specified earlier once a suffix tree has been created it can be used over and over again for searching purposes to retrieve certain patterns from the tree, which may include repeats.

Steller layout [24] was proposed by Bedathur and its main objective was to optimise disk based suffix tree from the search perspective. Steller's layout main contribution was to show how the balance between edge localisation and suffix tree localisation can be achieved. Their approach resulted in reduced I/O operations for search strategies involving suffix links. However, this approach did not consider any suffix tree property. Steller algorithm brings suffix tree links and tree edges together and this resulted in unnecessary localisation at some places, while at some locus it has to deal with lower localisation in comparison to other localisation of suffix tree nodes. The problem with Steller algorithm is that its arbitrary in nature even-though it enhances performance.

P Ko and Aluru's algorithm [25] theoretically gave a better disk layout for the suffix tree. The focus of their algorithm was to show how useful a secondary storage of a suffix tree (in solid state drives) is and how the suffix tree can be updated efficiently. This however is not the focus of this thesis because it is obvious that since DNA sequences are static in nature and they are very huge hence a secondary storage will be an ultimate option.

The clustering technique is not in any way related to the suffix trees but it provides a tree layout which gives a near optimal suffix-tree layout for exact match searches. Phoophakdee's algorithm showed that exact match alignment suffix-link version is better than non-suffix link version. The demonstration was only theoretical and was not proved experimentally. The algorithm proposed by Moffat [26] improves the search time after processing an already constructed suffix trees. LOF-SA algorithm [26] improved Moffat's algorithm giving it the ability to convert small tries structure to suffix arrays, hence this structure is outside our scope since this thesis focuses on improving the suffix tree not conversion of the suffix tree.

The other tree structure is the Compact Partitioned Suffix tree (CPS) [27], which reduces the number of page faults during search traversals. The CPS algorithm divides the suffix tree into various local trees, prioritizing nodes according to the maximum number of leaf nodes under it. This algorithm limits the worst case logical block access to $\theta(logm)$ where m is the number of characters in the indexed sequence, by implementing forward links. The scaling of CPS algorithm to a complete human genome has not been demonstrated. The search optimisation efforts by this algorithm are on the basis of exact matches, thus it avoids complex traversals patterns by not considering the suffix links.

4. Constructing a counter based suffix tree

The counter based suffix tree has been done but it involves constructing a normal suffix ukkonen suffix tree and then traversing the tree to label counters at each node. Hence, we propose a new suffix tree construction algorithm that creates counters at each node in one pass, which has not been outlined before. Assuming that we build a implicit suffix tree T from string $S\$$. The suffix tree is divided into $N-$phases and in phase $(i+1)$ tree T_{i+1} is constructed from T_i. Each phase has $i+1$ extensions, one for each of the $i+1$ suffixes of $S[1 \cdots i+1]$. During extension j of phase $i+1$, we locate the end of path from root labelled with substring $S[j \cdots i]$, then extends the substring by adding the character $S[i+1]$ to its end, unless its already appears there. Therefore in phase $i+1$, string $S[1 \cdots i+1]$, $S[2, \cdots i+1]$, $s[3, \cdots i+1]$ are added respectively (in extensions $1, 2, 3, \cdots$ respectively). Extension $i+1$ of phase $i+1$ extends the empty suffix of $S[1, \cdots i]$, meaning it puts a single character string $S(i+1)$ into the tree unless its already there. If $S[i+1]$ is already there in the tree it signifies that $S[i+1]$ has been repeated. Tree T_i is just a single edge labelled by character $S[1]$. The Algorithm 1 outlines the whole process.

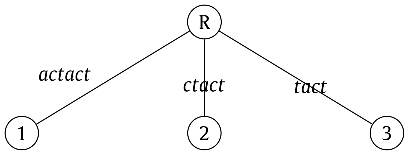

Fig. 1. An implicit suffix tree before the for *actact* before *g* is added.

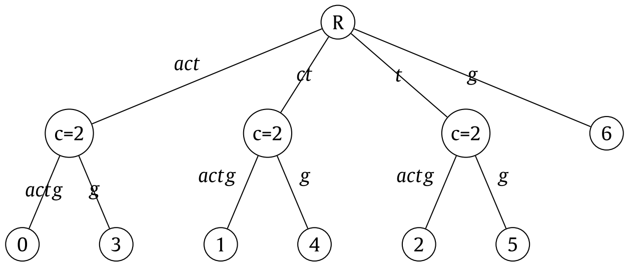

Fig. 2. The final suffix tree for *actactg* after the *g* was added.

4.1. Extension rules

To improve the Algorithm 1 we outline how to perform suffix extensions. Assuming $S[j\cdots i]=\alpha$ is a suffix of $S[1\cdots i]$, in extension j, when we find the end of α in the current tree. α has to be extended to insure $\alpha S(i+1)$ is in the tree. The following rules of extension must be observed with our **assertions** to make sure the counter based aspect is created:

- **Rule 1**: In the current tree, path α ends at a leaf i.e. the path from root labelled α extends to the end of some leaf edge. To update the tree, character $S[i+1]$ is added to the end of the label on the leaf edge.
- **Rule 2**: No path from the end of string α with character $S(i+1)$ but at least 1 labelled path continues from the end of α a new leaf edge must be created and labelled with $S(i+1)$ character. An internal node will be created if α ends inside the edge. The leaf at the end of the leaf edge is given the number j. We assert that *If an internal node is created a counter is generated and initialised to 2.*
- **Rule 3**: Some path from the end of the string α start with $S(i+1)$, and if $\alpha S(i+1)$ is already on the current tree we do nothing. However we assert that *If we skip an internal node and this rule applies that internal node counter is increased by 1.*

An example of an implicit suffix tree in Fig. 1 shows that it has only 3 suffixes because only rule 1 and 3 applied thats why we only have leaf nodes because when rule 3 we do nothing. Labels $1, 2$ and 3 are leaf nodes not internal nodes. Once g is added to the suffix tree all internal nodes are created and the counter initialised at 2 (i.e. $c=2$). As proposed earlier by our assertion to **rule 2** (see Fig. 2).

4.2. Algorithm speed-up and implementations

Using the rules only to build the suffix tree takes $\theta(n^3)$ where n is the string length. Gusfield [6] outlined ways to inhance the performance from $\theta(n^3)$ to $\theta(n)$. We therefore use the same techniques to prove that we can build the counter based suffix tree on such time complexity.

4.2.1. Suffix links

Definition. Given that $b\beta$ is an arbitrary string where b is a single character and β is a substring (possibly empty string). For any internal node v with path label $b\beta$ if there is another node $s(v)$ with path labelled β, then a pointer from v to $s(v)$ is a **suffix link**, however, if β is empty then the suffix link from the internal node with path labelled $b\beta$ goes to the root node.

Lemma 1. *If a new internal node v with path labelled $b\beta$ is added to the current tree in extension j of some phase $i+1$, then either the path labelled β already ends at an internal node of the current tree or an internal node at the end of string β will be created in extension $j+1$ in the same phase $i+1$. We **assert** that if the node already exists then the counter at that node is increased by one and if a new node is created that node counter will be equal or greater than v's.*

Proof. If and only if **rule 2** applies, then a new internal node is created in extension j, which means in extension j, the path labelled $b\beta$ is continued by some character other than $S(i+1)$ e.g. t. Therefore, in extension $j+1$ there is a path labelled β in a tree and it definitely has a continuation with character t (possibly with other characters). If β is continued by t only then rule 2 will create a node $s(\nu)$ at the end of path β and label the counter of this node to be the previous $nodecounter + 1$ or $suffixlinkcounter + 1$. When β is continued with other 2 different characters then there must be already a node $s(\nu)$ at the end of path $beta$ with a counter at some number which will then be increased by 1 and the Lemma is proved either way.

Corollary 1. *Any newly created internal node will have a suffix link from it by the end of the next extension and a counter, if a new internal node is created its counter will never be less than that of the suffix link.*

In reference to Corollary 1 all internal nodes with counters in the changing tree will have suffix links from them except the most recently added internal node which will receive its suffix link by the end of the next extension bearing in mind that a new node created via rule 2 extension has a counter initialised to 2.

We submit that the use of suffix link as in Ukkonen's algorithm can be used preserving the node counters and as thus improves the worst case running time from $\theta(n^3)$ to $\theta(n^2)$. Walking down the tree using suffix links is also paramount to developing a counter based suffix tree.

4.2.2. *Skip/count trick*

In an effort to reduce the traversal time to something proportional to the number of nodes on a particular path, we will use the skip/count trick.

Trick 1: To illustrate let r denote the length of γ, and as we recall; no two labels of edges out of $s(\nu)$ can start with the same character, the first character of γ must appear as the first character on exactly one edge out of $s(\nu)$. Let $r\prime$ denote the number of characters on that edge. If $r\prime$ is less than r then we do not have to look at any more of the characters on that edge; we simply skip to the node at the end. Then r is set to $r - r\prime$, variable h to $r\prime + 1$ and looks at the outgoing edges to find the current next edge (whose character matches with h of γ). Generally we identify the next edge on the path and compares the current value of r to the number of characters $r\prime$ on that edge, and if r is at-least as large as $r\prime$ we skip to the node at the end of the edge, set r to $r - r\prime$, h to $h + r\prime$, then finds the edge whose first character is r of γ and repeats. However when the edge is reached, and r is smaller than or equals to $r\prime$, then we skip the r to the edge and quit, having achieved the fact that γ path from $s(\nu)$ ends on that edge exactly r characters down its label. By so doing the total time to traverse the path is then proportional to the number of nodes on it rather than the number of characters on it. We assert that when constructing the counter based suffix tree this trick will reduce the time complexity at each phase to $\theta(n)$ just as in the normal ukkonen algorithm.

However the whole algorithm implemented with suffix link will still take $\theta(n^2)$, which is achieved by multiplying the time complexity of each phase by n because there are a n phases.

4.2.3. *Edge label compression*

Edge label compression is an effort to reduce the space needed to store the characters and instead of storing the characters on each edge we will store the pairs of indices and the copy of the actual string as well as the counter value of each node. Storing index pairs means only 2 numbers are written on any edge and since the number of edges is at most $2n - 1$ the suffix tree uses only $\theta(n)$ symbols, hence it requires only $\theta(n)$ for Ukkonen whereas in our case we will need more space to store counters at each internal node thereby increasing the space complexity.

4.2.4. *Stop if rule 3 applies*

There is an observation that rule 3 is a show stopper in any phase if rule 3 applies in extension j, it will also apply in all further extensions ($j + 1$ to $i + 1$) until the end of the phase. This is because if rule 3 is applied the path labelled $S[j \cdots i]$ in the current tree must continue with characters $s(i + 1)$, therefore the path labelled $S[j + i] \cdots i$ does also, and it also applies in extensions $j + 1, j + 2, \cdots i + 1$.

We **assert** to this observation that if this rule applies it indicates repetition of some sort, therefore, if rule 3 applies past an internal node its counter is increased by 1. A new suffix link needs to be added after the extension in which rule 2 applies.

Trick 2: When rule three applies we check if it has passed an internal node, if that is the case we increase node counter by one and phase is ended.

4.2.5. *Once a leaf always a leaf*

The trick here is basically to represent the current end with a global end for all leafs. This means that once rule 1 happens we do not change anything because as since it's a global end and in each phase global end will be increased by 1.

Trick 3: In phase $i + 1$ when a leaf edge is created it will normally be labelled with string $S[p \cdot \cdot i + 1]$, instead of indices $(p, i + 1)$ on the edges we write p, e, where e is a symbol denoting "current end". e is the *global* index, that is set to $i + 1$ once in each phase. During phase $i + 1$ we know rule 1 will apply in extension i through j, by so doing we cut the additional work of implementing unnecessary j_i extensions. Instead e is constantly increased but then explicit work for some extensions starting with extension $j_i + 1$. This trick will come in handy for our counter based tree as internal nodes are not affected by these rule hence no adjustments are made to this trick.

Algorithm 2 Counter Based suffix tree implementation.

```
1: $root \leftarrow SuffixNode$
2: $rootindex \leftarrow -1$
3: $active \leftarrow rootnode$
4: $e \leftarrow -1$
5: for $i \leftarrow 0$ to $length(S)$ do
6:     Begin Phase$(i)$
7: end for
```

Algorithm 3 Single Phase extension.

```
1: $e \leftarrow e+1$
2: $lastCreatedNode \leftarrow null$
3: $r \leftarrow r+1$
4: while $(r > 0)$ do
5:     if $Activelength \leftarrow 0$ then
6:         if currentch from root $\neq null$ then
7:             $Activeedge$ ←currentch from root
8:             $activelength \leftarrow activelenth + 1$
9:         else
10:            Create $Suffixnode$
11:            $Suffixnodecounter = 2$, $r = r - 1$
12:        end if
13:    else
14:        if currentch == nextch && $lastCreatedNode \neq null$ then
15:            $lastCreatedSuffixlink \leftarrow currentNode$
16:        end if
17:        walkdown past the node and increment $nodecounter$ by 1.
18:        Break out of the loop.
19:    end if
20:    Create a new $internalnode$
21:    $nodeCounter \leftarrow 2$, Create new leaf node
22:    Set internal nodes child as old node and new leaf node.
23:    if $lastCreatedInternalNode \neq null$ then
24:        $lastCreatedInternalNode\ suffixlink \leftarrow newInternalNode$
25:    else
26:        $lastCreatedInternalNode \leftarrow newInternalNode$
27:        $newInternalnodecounter ++$
28:        $NewinternalNode\ suffixlink \leftarrow root$
29:    end if
30:    If active node is not root then follow suffix link
31:    If active node is root then increase active index by one and decrease active length by 1
32:    $e \leftarrow e - 1$
33: end while
```

Corollary 2. *If a newly created node has a suffix link which is not root, then its counter will be 1 more than the $suffix - link$ counter.*

Proof. To have visited a suffix link it means that there has been a repeat therefore making sure that counter of the newly created is never greater that the suffix links node counter insures that counters for shorter strings are correct.

Lastly we consider the active points in our counter based suffix tree construction. These are the guides to when the next extension or phase will start. We outline rules and their effects to active points. **Active Edge** is the index of the actual character in the string, **Active length** is a placeholder that helps trace how far we are from the active edge. **Active node** the node from which an active point will start. If active length is 0, the next phase or extension will start at the *root*. Its either the extension follows from the internal node or the root node. When rule 3 happens the active length and edge will be increased by 1, however if jumping across the internal node its counter will be increased by 1 and active node is reset to that node. The active edge will then became the character index $S[i+1]$ and the active length is reset to 1 and increases accordingly. If rule 3 happens we create a path and reduce active edge by 1 and increase active length by 1. If active node is not at the root we follow the suffix links to reach the next node and reaching that node increases that node counter by 1. The whole process is summarised by the following Algorithm 2.

Algorithm 3 outlines a step by step construction of a single phase extension. This construction makes use of the active points. Following the these algorithms Fig. 3 is created.

5. Searching for repeats

Once the suffix tree has been created it can be searched over and over again to extract patterns. This makes more sense if the suffix tree is constructed for a human DNA genome because the DNA of an individual stays the same hence there is no need to always construct the suffix tree every time a search is to be performed in an individual's DNA. Producing an efficient suffix tree for a genome becomes an essential part in bio-informatics. Throughout this thesis the discussions have

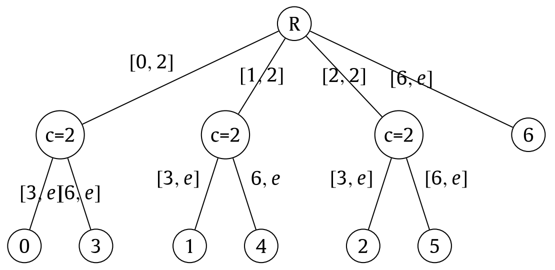

Fig. 3. The final suffix tree for *actactg* after the *g* was added.

Table 1
The execution time taken by the algorithm that construct the Ukkonen suffix tree and the execution time taken by the proposed algorithm that constructs the Counter Based suffix tree.

Length of the string	Ukkonen suffix tree	Counter based suffix tree
10 000 000	191	221
11 000 000	242	275
12 000 000	342	378
13 000 000	734	751
14 000 000	1024	1043
14 250 000	1204	1240

been about producing a suffix tree that will help in quickly identifying repeats from a suffix tree. With the original suffix tree finding the occurrences of substring q in S involves following the path for q from root and then counting the number of leaves under the node at which substring q ends will therefore be the number of occurrences of substring q. This process will involve searching beyond the substring to be checked hence increases the search time over the suffix tree. If the suffix tree is constructed with node counters then the search will involve following the substring q and when reaching the end of the path check what the counter reads and then records that as the number of occurrences of substring q. With this suffix tree data structure much effort is reduced because there is no need to search beyond the path of the substring and this will theoretically reduce the search time of substring q.

One question that needs to be addressed is whether this data structure will in any way hinder the general applications of a suffix tree besides searching for repeated sequences. The other application of the suffix tree is to aid the finding for longest repeats in a sequence. The normal suffix tree will prompt the user to find the deepest node that has at least 2 leaves under it. With the suffix tree that has node counters the search will be done the same way but there won't be any need to check for leaves under the deepest node only to check if the counter at that node has a value more than 1.

6. Results and discussions

In this section the experiments were done along with their results. The results were done based on the time each algorithm will take to perform each task on the data that is availed to the algorithm. The tasks that the algorithm must do is create a suffix tree and then search for repeats on the created suffix tree which will be stored in the main memory of the computer. The alphabets that are supposed to be used are $\{a, c, t, g\}$ which represents the nucleobases, however since the Human Genome will have an extra character n, which represents the confidence of the alleged base the algorithm must accommodate for such instances. This means that the character set that the experiments will be based on are $\{a, c, t, g, \$$ and $n\}$. $ is the unique character that does not appear anywhere in the entire genome and it will be appended to each cut-off string to construct the suffix trees.

To efficiently measure the performance of the two algorithms coding of these two algorithms was done on the RIVYERA S6-LX150. This machine has the FPGA processing capabilities however this capability was not utilized in this thesis. The working memory (RAM) that was used to run these programs is 32GB. The system made use of the Linux server and the SSH to connect remotely to the Linux server in the machine is PUTTY. Coding the algorithm was done in python 3.4 (in appendix 2) which was installed in the server.

The experiment of construction a suffix tree was done in two parts. The first experiment was based on the Ukkonen suffix tree construction algorithm which was proposed by Dan Gusfield [6]. To record the time taken to construct the suffix tree for varying lengths, the execution time was recorded using the python build in function timeit, which will record the time in seconds it takes for a construction of Ukkonen suffix tree. And the results was recorded in the second column of Table 1.

The second experiment was based on the counter based suffix tree. The time taken for the algorithm to construct a counter based suffix tree was done using the *timeit* python function as well using the same varying length of strings as in the first experiment. During this experiment the minimum number of rule 3 extension was set to 40 characters to avoid

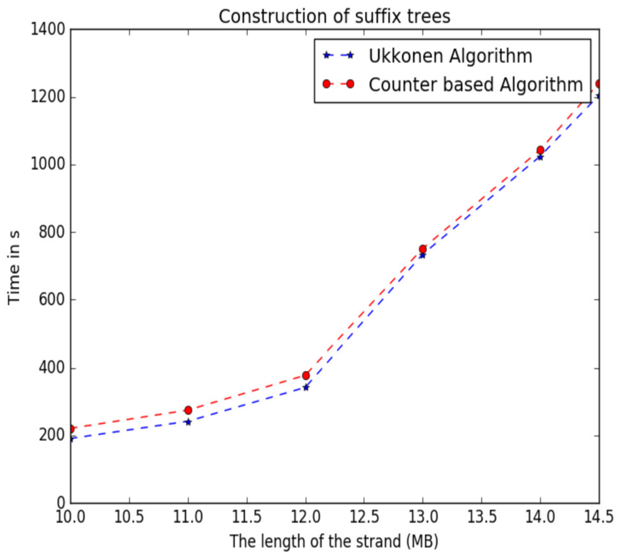

Fig. 4. Graph representation of the time taken for the algorithm proposed by Ukkonen to construct a suffix tree as opposed to a proposed Counter based suffix tree algorithm.

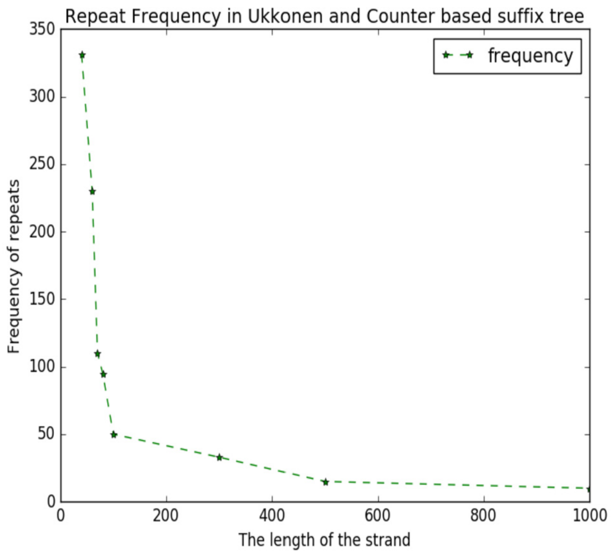

Fig. 5. Graph representation of the frequency of repeats found by both the counter based and the Ukkonen suffix tree.

adding counters to repeats that will be too short. The results were then recorded in column 3 of Table 1. In Fig. 4, the results of Table 1 were presented as a graph to depict the execution time of these two experiments.

The suffix tree construction of a counter based suffix tree takes a noticeably longer time to construct and the reason for that is that the algorithm has to be delayed by tagging the possible repeated nodes with counter and this action delays the time complexity. However the good part about the counter based suffix tree is that, although the ukkonen algorithm out performs the counter based suffix tree the difference is very small. The average time in which the Ukkonen suffix tree outperforms the counter based suffix tree during construction is merely 31.5 seconds which is significantly small.

6.1. *Searching for repeats*

The Ukkonen constructed algorithm is outperformed by the counter based suffix tree when it comes to searching for repeats. In this section of thesis we will explore the ability of the counter based algorithm in searching for repeats. The first test case used is providing different string of varying sizes and search them in both the ukkonen and the counter based algorithm. And the results were plotted in the Fig. 5.

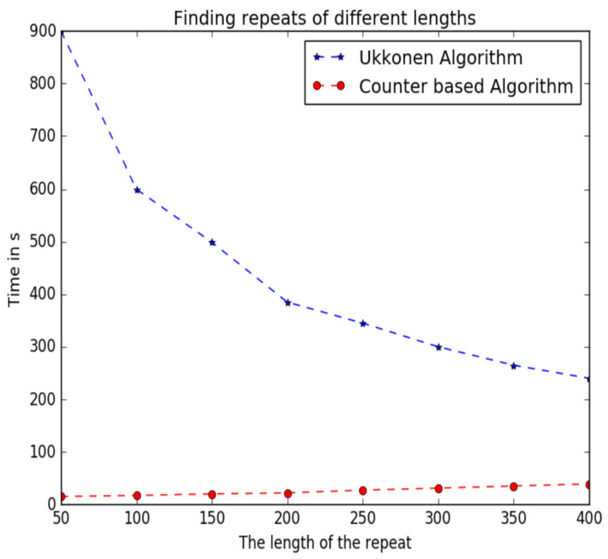

Fig. 6. Graph representation of the time taken for finding repeats of different lengths.

And both algorithms were able to find the exact repeats from the respective suffixes tree and the and the results were displayed. The results of this experiment gave the expected results since both the Ukkonen and the Counter based algorithm must give the consistent results but at different speed. The suffix tree constructed was of character length of 1 425 000 neucleic bases including the no confidence bases. Fig. 5 shows the total frequency of repeats of different string sizes.

When making use of the suffix tree to aid repeat finding the shorter exact repeat will be more frequent than the longer repeats [10]. Fig. 5 highlights this claim this is very true for finding exact repeats as opposed to tandem repeats or approximate repeats.

The next important aspect to consider was the time each algorithm takes to perform the search time for repeats. To investigate this test case a suffix tree of length 1 425 000 characters was constructed for both the counter based algorithm and the traditional Ukkonen algorithm. A known repeat from the dataset of different length was searched and the time taken was recorded in the Fig. 6.

As explained in Section 4 earlier when searching for repeats the in the ukkonen suffix tree, the strategy is to search beyond the string to be searched and then count the number of leafs from that point which means that the search time will be increased significantly for short sequences. And as we increase the length of the sequences the search effort will also reduce in the ukkonen algorithm as the need to find repeats will be reduced. Fig. 5 shows how the repeat search time of the ukkonen algorithm will behave as the length of the string is increased. When searching for these repeats in the counter based suffix tree there is an interesting slight increase as the repeat length is increased. The reason for this is simple when searching for shorter repeats of the algorithm will simply follow the path of the repeat and when it reaches that point it checks what the counter is at that point and then it reports the counter reading as the frequency of repeats. The search effort of the counter based algorithm is reduced by a large margin for shorter strings as opposed to the ukkonen suffix tree which does more work of traversing beyond the point of interest. There is a noticeable increase as the string length increase the reason for this is that the traversal of the string is also increasing as the string length increases hence more effort is understandably required.

7. Conclusions

This paper presented the construction of a counter based suffix tree in contrast with the traditional Ukkonen algorithm. The proposed counter based suffix tree proved to have the on-line construction property that is inhibited by the traditional suffix tree. Although the construction of this suffix tree is on-line its main disadvantage is the space complexity needed to store counters together with the internal nodes. Findings of this thesis proved that in terms of time efficiency the Ukkonen algorithm outperforms the counter based suffix tree, because of the time needed to tag each path if **rule 3** extension applies. However in relation to finding the repeats the counter based suffix tree proved to be very useful since the effort to search for repeats on the suffix tree constructed that way demands less and less efforts. Fig. 6 in results and discussions shows a remarkable and noteworthy difference. When finding repeats of smaller lengths, the Ukkonen suffix tree search strategy will spend more searching beyond the needed string which will result in more search time whereas the counter based suffix tree tends to follow the path of the string repeated and when it reaches that point it reports the counter reading as the repeat at that particular point. The counter based suffix tree is a better way to construct a suffix tree for repeat searches.

8. Recommendations and future work

This paper outlines the construction of the suffix tree on a cutoff bases from the dataset which is approximately 5GB and only 14.25MB was used to demonstrate the practicality of constructing a counter based suffix tree. The base pairs involved during this process was not as near close to the X-chromosome because the construction of the suffix tree was done in the primary memory and the tree was stored there, therefore the machine used could not allow for us to build a suffix tree for 5GB string dataset. Hence for the construction of a large dataset of this magnitude we need to construct the suffix tree and store it in a secondary memory. There are lot of methods explained in this thesis on how a suffix tree can be stores in a secondary memory efficiently. These researches [17] [22] [23] [19] uses the concept of a traditional ukkonen algorithm and some changes to the concept of suffix links. In the future, exploration on how these researches can be made relevant to the counter based suffix tree will be an interesting avenue to investigate. A counter based suffix tree tends to give very large counter values to short repeats during construction which will increase the space complexity of the algorithm therefore setting the $minlenR$ will be an excellent idea to reduce the space complexity.

A lot of algorithms [8] [11] [24] [7] have been developed in order to efficiently search for repeats in Ukkonen suffix tree, it might be interesting to see if any of these techniques might work on the counter based suffix tree which in actual fact is an enhanced version of the ukkonen suffix tree. Still on the concept of finding repeats it is very crucial that the algorithm have the ability to find not only the exact repeat but also the appropriate repeats.

The construction of a suffix tree was done on a machine that has FPGA processing capabilities, hence it supports parallel programming to reduce the computation time. [28] has outlined the construction of Ukkonen algorithm in parallel programming. Counter based suffix tree can be build on the dataset, hence investigating how generalized can be implemented with counters and how merging these trees together will be an interesting avenue to explore in the future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J.P. Noonan, Neanderthal genomics and the evolution of modern humans, Genome Res. 20 (5) (2010) 547–553.

[2] K. Shoji, K. Oda, S. Nakagawa, S. Hosokawa, G. Nagae, Y. Uehara, K. Sone, Y. Miyamoto, H. Hiraike, O. Hiraike-Wada, et al., The oncogenic mutation in the pleckstrin homology domain of akt1 in endometrial carcinomas, Br. J. Cancer 101 (1) (2009) 145–148.

[3] A.J. Jeffreys, V. Wilson, S.L. Thein, et al., Hypervariable 'minisatellite' regions in human dna, Nature 314 (6006) (1985) 67–73.

[4] N. Gilbert, J. Allan, Distinctive higher-order chromatin structure at mammalian centromeres, Proc. Natl. Acad. Sci. 98 (21) (2001) 11949–11954.

[5] J. Zheng, S. Lonardi, Discovery of repetitive patterns in dna with accurate boundaries, in: Fifth IEEE Symposium on Bioinformatics and Bioengineering, BIBE'05, IEEE, 2005, pp. 105–112.

[6] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[7] A.F. Smit, R. Hubley, P. Green, Repeatmasker, Published on the web at http://www.repeatmasker.org, 1996.

[8] C. Felicioli, R. Marangoni, Bpmatch: an efficient algorithm for a segmental analysis of genomic sequences, IEEE/ACM Trans. Comput. Biol. Bioinform. 9 (4) (2012) 1120–1127.

[9] A.L. Price, N.C. Jones, P.A. Pevzner, De novo identification of repeat families in large genomes, Bioinformatics 21 (suppl 1) (2005), i351–i358.

[10] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, R. Giegerich, Reputer: the manifold applications of repeat analysis on a genomic scale, Nucleic Acids Res. 29 (22) (2001) 4633–4642.

[11] D. He, Using suffix tree to discover complex repetitive patterns in dna sequences, in: Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE, IEEE, 2006, pp. 3474–3477.

[12] H.H.-W.W. Xiao-Wu, An adaptive suffix tree based algorithm for repeats identification in a dna sequence, Chinese J. Comput. 4 (2010) 011.

[13] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[14] Y. Peng, H.C. Leung, S.-M. Yiu, F.Y. Chin, Idba–a practical iterative de bruijn graph de novo assembler, in: Annual International Conference on Research in Computational Molecular Biology, Springer, 2010, pp. 426–440.

[15] Y. Tian, S. Tata, R.A. Hankins, J.M. Patel, Practical methods for constructing suffix trees, VLDB J. 14 (3) (2005) 281–299.

[16] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.

[17] S.J. Bedathur, J.R. Haritsa, Engineering a fast online persistent suffix tree construction, in: Data Engineering, 2004. Proceedings. 20th International Conference on, IEEE, 2004, pp. 720–731.

[18] S. Tata, R.A. Hankins, J.M. Patel, Practical suffix tree construction, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, VLDB Endowment, 2004, pp. 36–47.

[19] B. Phoophakdee, M.J. Zaki, Genome-scale disk-based suffix tree indexing, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, 2007, pp. 833–844.

[20] B. Phoophakdee, M.J. Zaki, Trellis+: an Effective Approach for Indexing Genome-Scale Sequences Using Suffix Trees, Pacific Symposium on Biocomputing, vol. 13, Citeseer, 2008, pp. 90–101.

[21] M. Barsky, U. Stege, A. Thomo, C. Upton, A new method for indexing genomes using on-disk suffix trees, in: Proceedings of the 17th ACM Conference on Information and Knowledge Management, ACM, 2008, pp. 649–658.

[22] M. Barsky, U. Stege, A. Thomo, C. Upton, Suffix trees for very large genomic sequences, in: Proceedings of the 18th ACM Conference on Information and Knowledge Management, ACM, 2009, pp. 1417–1420.

[23] A. Ghoting, K. Makarychev, Serial and parallel methods for i/o efficient suffix tree construction, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, ACM, 2009, pp. 827–840.

[24] S.J. Bedathur, J.R. Haritsa, Search-optimized suffix-tree storage for biological applications, in: International Conference on High-Performance Computing, Springer, 2005, pp. 29–39.

[25] P. Ko, S. Aluru, Obtaining provably good performance from suffix trees in secondary storage, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2006, pp. 72–83.
[26] R. Sinha, S. Puglisi, A. Moffat, A. Turpin, Improving suffix array locality for fast pattern matching on disk, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, 2008, pp. 661–672.
[27] S.-S. Wong, W.-K. Sung, L. Wong, Cps-tree: a compact partitioned suffix tree for disk-based indexing on large genome sequences, in: 2007 IEEE 23rd International Conference on Data Engineering, IEEE, 2007, pp. 1350–1354.
[28] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, U. Vishkin, Parallel construction of a suffix tree with applications, Algorithmica 3 (1–4) (1988) 347–365.