The Adaptive Suffix Tree: A Space Efficient Sequence Learning Algorithm

Upuli Gunasinghe and Damminda Alahakoon

Abstract— The Adaptive Suffix Trie algorithm was previously proposed by the authors as a sequence learning algorithm for capturing frequent sub sequences of variable length. This algorithm builds up a suffix trie data structure, capturing repetitive patterns in a given set of sequences. Its application has been demonstrated in bioinformatics and text clustering. Suffix trees are the space efficient variant of suffix tries and are thus more widely used in the current literature. In this paper we propose the Adaptive Suffix Tree algorithm, which is based on the same learning principles as the Adaptive Suffix Trie, but has the advantage that it is more space efficient. We discuss the new algorithm in detail and demonstrate that the same set of sub sequences can be learnt by the proposed algorithm while utilizing less than 50% of the space used by its predecessor.

I. INTRODUCTION

SEQUENCES occur naturally in most cognition based activities carried out by humans as well as in a diverse set of domains that we work in [1],[2]. Sequences can be of varying length and have hierarchical structure. For example in speech, words are combined to form phrases, phrases join to form sentences and sentences are used together in conveying meaningful messages. Motor control or the actions we perform are carried out by sequentially coordinating our muscles. Critical thinking and problem solving also involves analysing information over a period of time and coming up with a solution.

In addition, most domains consist of information that is sequentially structured. For example information coded as DNA sequences or RNA sequences are found in biology [3]. Time based applications in industries such as health, aviation and utilities etc. capture sequential information of temporal nature that require analysis or monitoring. Further, masses of sequential information are stored in the form of text such as in books, journals, websites etc.

Many methods and techniques have been proposed to analyse these masses of sequential information. Some of the early neural network based approaches simply converted the serial information into a parallel space, processing an entire sequence as a single static input [4], [5]. However, due to the increase in dimensions and the need for extra buffering space and techniques to store the entire sequence at once, these approaches were not efficient in terms of time and space. Another approach was to add memory to neural networks such that they remembered the previous state [6], [7], [2]. Although these networks performed well for shorter sequences they have difficulty in handling sequences that span longer intervals [8].

Hidden Markov Models are probability based models which are mainly used in speech recognition and bioinformatics as well as in a wide range of other applications [9]. However, the Markovian assumption makes them inappropriate for dealing with long sequences where dependencies might span several consecutive states. In addition, many self organizing map (SOM) [10] based solutions for temporal sequences have been proposed such as the Temporal Kohonen Map (TKM) [11] and the recursive self organizing map (RecSOM) [12]. These work well with short sequences. However, they cannot deal with variable length sequences. The Merge SOM [13] which was introduced later has the ability to deal with variable length sequences. However, due to the learning technique applied, it has the limitation of being more biased towards the last elements of long sequences.

One common problem in the above solutions is that they face difficulties when dealing with long sequences. In addition, some solutions cannot work with variable length sequences. In [14] we introduced the Adaptive Suffix Trie (ASTrie), a sequence learning algorithm for capturing long variable length repetitive patterns. The ASTrie is based on the Hebbian learning rule [15]. Using the Hebb rule, it builds up a suffix trie data structure capturing only the frequent sub sequences in a given set of sub sequences. We demonstrated how the algorithms learning parameters can be selected by the user in order to learn sequences with the required characteristics. The algorithm has been applied in two different domains. First in finding evolutionary relationships between long dense sequences in biology [14]. Then in the text domain, for clustering of short sparse sequences [16].

In this paper we present the Adaptive Suffix Tree (ASTree) algorithm, as a space efficient successor of the ASTrie. When compared to the suffix trie, the suffix tree is a data structure that utilises much less space. Therefore, the ASTree algorithm we present in this paper is the compressed version of the ASTrie algorithm. It works on the same principles as the ASTrie algorithm, capturing the same number of sub sequences. However, it utilizes much less space when compared to its predecessor.

The paper is structured as follows. In section II we give an overview of suffix data structures and discuss the ASTrie algorithm. We then introduce the novel ASTree algorithm and discuss its learning parameters and functions in section III. Experiments and results comparing the ASTrie and ASTree algorithms in terms of space and time are described in section IV. We conclude the paper by discussing the implications of the presented work and future improvements.

Upuli Gunasinghe is with the Faculty of Information Technology, Monash University, Australia and Damminda Alahakoon is with the School of Information Systems, Deakin University, Australia (email: upuli.gunasinghe@monash.edu, d.alahakoon@deakin.edu.au).
II. BACKGROUND

In this section we discuss the literature related to the ASTrie algorithm. We give an overview of suffix data structures and discuss the implementation of the ASTrie algorithm.

A. Suffix data structures

Tries, suffix tries and suffix trees are tree data structures which are used widely for solving string based problems [17], [18]. In a trie [18] each edge is labelled with a single character. Each path from root to a leaf corresponds to a string represented by a trie. Words with common prefixes can be identified by looking for paths that branch off from the same internal node. An example of a trie for the words bat, ball, cub and club is given in figure 1.

![A trie for the strings bat, ball, cub and club](image1)

A suffix trie [18] is a trie which consists of the suffixes of a string. For example consider the DNA sequence fragment AACACGTACG. The suffixes of this string would be AACACGTACG, ACACGTACG, CACGTACG, ACACGTAGS, CACGTAGS, GTACG, TAGS, ACG, CG and G. A $ character is added to the end of the string to identify each suffix clearly in the suffix trie. The suffix trie would represent all these suffixes in its structure as shown in figure 2.

![A suffix trie for the string AACACGTACG](image2)

A suffix tree [17] is a compressed suffix trie, where all internal nodes have at least two children. The paths in a suffix trie which consist of nodes that have a single child are compressed into a single edge. The characters of this path are mapped to the single edge representing them. The suffix tree corresponding to the string AACACGTACG is given in figure 3. It can be seen that the suffix tree is much more efficient in terms of space when compared to suffix trie. Many time efficient algorithms for constructing suffix trees have been introduced [17]. In addition, suffix arrays have been proposed as space efficient versions of suffix trees and these have the ability to perform most of the operations that are carried out by suffix trees.

![A suffix tree for the string AACACGTACG](image3)

For a set of strings, generalized suffix structures can be formed by appending a different character to the end of each string in order to identify its suffixes separately [17]. When compared to suffix tries, suffix trees are more widely used for solving problems in stringology since they are space efficient. However, suffix trees still have the problem of consuming a large amount of space because they build up the entire tree structure for all the suffixes in a sequence. Many space efficient suffix tree building algorithms have been proposed [19], [20]. In addition, sparse suffix trees which represent only a subset of suffixes [21] and truncated suffix trees which have a limitation on the height of the suffix tree [22] have been introduced. These solutions cater for stringology based applications such as the longest common substring problem, the longest prefix repeat problem and data compression etc. [17]. Although it might be necessary to build the entire suffix tree for string problems, for most data mining applications based on sequences it is not necessary to look at all sub sequences. Rather a subset of interesting sub sequences which occur frequently are of more interest. Sub sequences which occur once or twice could be disregarded as outlier information. In addition, depending on the application it might be required to control the length and frequency of the sequences being learned.

The ASTrie [14] was proposed as a solution to this problem. It has the ability to learn variable length frequently occurring sub sequences as per user requirement. We use the term sub sequence to represent a consecutive string of elements in a given sequence. We use the phrase sub sequence in place of substring, since the ASTrie has not only been used in capturing substrings but for learning variable length phrases in text clustering applications. Below we discuss the ASTrie algorithm in detail.

B. ASTrie algorithm

The ASTrie builds up a generalized suffix trie for a given set of sequences. However, it does not store all the suffixes in the sequences. Rather it stores only the most frequently occurring sub sequences. This is achieved by using a learning rule based on Hebbian learning [15]. The Hebb rule has been used in many learning algorithms [10], [23], [24]. It is a biologically inspired learning rule and explains how learning occurs in higher vertebrates. The Hebb rule is described as follows: “When an axon of cell A is near enough to excite
a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that As efficiency, as one of the cells firing B, is increased” [15]. Both the ASTree and ASTrie includes a weight adaptation process based on the Hebb’s rule such that if a element A follows element B repeatedly in a set of sequences, the weight of the object representing B will be adapted towards the weight of the object representing A.

Each node in the ASTrie keeps track of the character or element which corresponds to the edge between it and its parent. Each node also has a weight and a state assigned to it. The state of a node can be either mature or non-mature. The root node is created as a mature node with a weight of 0. All other nodes at time of creation are non-mature and assigned a weight of 1. Child nodes can only be added to mature nodes. When a sub sequence corresponding to the path from the root to a node, appears in the input sequence, the node’s weight is adapted towards the weight of its parent. When the weight difference between the parent and the node is less than a predefined threshold value, the node’s state is set to mature. It is also assigned the weight of its parent. New nodes can now be added to it since it is mature. After training, all non-mature nodes are pruned from the tree.

The node’s mature property is set to true. Since the root is created as mature, it’s mature property is considered to be true.

The ASTree algorithm includes a weight adaptation process based on the Hebb’s rule such that if a element A follows element B repeatedly in a set of sequences, the weight of the object representing B will be adapted towards the weight of the object representing A.

Below we introduce the new Adaptive Suffix Tree algorithm for learning variable length sequences. It is an improvement to the ASTrie in that it uses much less storage space but can capture the same amount of sub sequences.

### III. Methods

In this section we explain the novel ASTree algorithm in detail. We discuss the algorithm providing graphical examples of the tree’s growth and give a detailed explanation of its learning parameters and learning rate functions.

The ASTree algorithm builds up a generalized suffix tree, capturing only the frequent sub sequences present in a set of sequences. Similar to the ASTrie it learns the common sub sequences through weight adaptation using the Hebbian learning principle. The key difference is that, each node stores a list of elements instead of a single element that corresponds to the elements on the edge between the node and its parent. The ASTree algorithm incorporates this difference in its implementation. Below we discuss the algorithm in detail.

#### A. The ASTree algorithm

The ASTree algorithm consists of two phases. The initialization phase and the learning phase. It is an on-line algorithm that processes each element of a sequence as it is presented, requiring no prior knowledge of the sequences and no additional memory space for input storage. A suffix tree is constructed based on the sequences that have been processed. Nodes in the ASTree have three parameters, the weight, learning rate and mature property. The weight and the learning rate can take a value between 0 and 1 while the mature property is a binary variable taking the values of true or false. Since a node stores a list of elements, if the mature property is set to false, it only affects the last element of the node. For all other elements, the node’s mature property is considered to be true.

During the initialization phase of the algorithm, the root of the ASTree is created. Its weight is set to 0 and its mature property is set to true. Since the root is created as mature, learning is not required to take place and child nodes can straight away be added to the root. The algorithm keeps a list of the root and mature nodes which have been accessed in the previous state in memory. Items in this list consist of <node,count> value pairs where the count represents the position of the previously processed element of the sequence in the list of elements in the node. In the initialization phase this list is created and the <root,0> is added to it.

Based on the learning rate function being used, all nodes in the tree could have the same learning rate or each node could have a different learning rate that depends on its depth. In the latter case all nodes of the same depth would have the same learning rate associated with them. These learning rates are pre-calculated and stored in memory during the initialization phase. A detailed discussion regarding the learning rates is discussed below in section III-C.

The learning phase comes next. Here, each sequence is presented to the algorithm. Given a sequence, the algorithm

![Fig. 4. The growth of the ASTrie for the string AACACGTACG](image-url)
processes each element in it sequentially. When an element is processed, the paths corresponding to the partial suffixes representing the element is traced down the tree. This can be done by accessing the list which holds the root and the previously accessed nodes and checking if the next element or child of the elements in the list corresponds to the current element. If a node corresponding to the current element exists, and the node is not mature, then the weight of the node is adapted towards the weight of the node’s parent.

If a node corresponding to the current element does not exist, changes are made to represent it in the tree. This is done by either (a) an element corresponding to the current element is added to the list of elements of the node representing the previous element; or (b) a new node corresponding to the current element is created and added as a child to the node representing the previous element; or (c) the node representing the previous element is split.

(a) is carried out if the node representing the previous element is a mature node which has no children, and the previous element is the last element in its list of elements. This process is shown in figure 5. The node $n_2$ is a mature node which has no children holding the previous element $S_i$. The current symbol $S_{i+1}$ is appended to the end of $S_i$. The node $n_i$ is set to non-mature and its weight is set to 1. A weight adaptation is then carried out for node $n_i$ depending on the learning parameters.

(b) is carried out if the node representing the previous element is a mature node which has children, and the previous element is also the last element in its list of elements. This is illustrated in figure 6. Node $n_i$ is the mature node holding the previous element $S_i$. However, it has other children which do not correspond to the current element. Therefore, a new node $n_{i+1}$ is created and added as a child of node $n_i$. The new node’s weight is then adapted towards $n_i$.

(c) is carried out if the previous element is an internal element in the list of elements represented by the node representing it. Here a splitting process of the node corresponding to the previous symbol $n_i$ takes place. Two new nodes $n_{i1}$ and $n_{i2}$ are created, $n_{i1}$ with the elements after the previous symbol of node $n_i$ and $n_{i2}$ with the current element. The weight and mature property $n_i$ is transferred to $n_{i1}$. If $n_i$ has any children they are removed from $n_i$ and added to $n_{i1}$. The weight of $n_i$ is set to 0 and its mature property is set to true and the symbols after the previous element are removed from its list of symbols. Both $n_{i1}$ and $n_{i2}$, the newly created nodes are added as children to $n_i$. Finally node $n_{i2}$’s weight is adapted towards the weight of node $n_i$. This process is illustrated in figure 7.

An example of how the ASTree is grown for the input string AACACGTACG is shown in figure 8. During initialization the root node is created as a mature node with weight set to 0. When the symbol A is input, since the root does not have any children, it is attached to the root node and the root node’s weight is set to 1 and mature property to false. A weight adaptation also takes place. Since the root does not have a parent, the adaptation is made towards a value of 0. The weight is again adapted when the next A is presented and the node becomes mature. When the next symbol C is input, the list of previous elements would take the form of ($<\text{root},0\text{>,}<\text{root},1\text{>}$). When processing the first element in the list, the root node is split, such that it has two child nodes, $n_1$ and $n_2$. The symbol C is then appended to the end of node $n_1$ holding A, resulting in the tree given in figure 8 (c). When the next two symbols A and C are input, nodes $n_1$ and $n_2$ reach maturity. The next symbol G is appended to the nodes, $n_1$ and $n_2$ and a new symbol with the element G is also created. These three nodes reach maturity and a new node $n_4$ for T is added as the next set of symbols are processed as shown in figure 8 (f)-(g). Finally the non-matured node T is removed from the tree.

When compared to the ASTrie for the same input shown in figure 4, it can be seen that the ASTree has a reduction of nearly half the nodes. However, all the information in the ASTrie is captured in the ASTree. Therefore, the ASTree has a huge advantage over space when compared to the ASTrie.
algorithm. The algorithm for constructing the ASTree for a given set of sequences is provided in Algorithm 1. Below we discuss the growth of the tree in terms of weight adaptation and learning rate functions.

B. Weight adaptation

If a node corresponding to the current element is non-mature, its weight is adapted towards the weight of its parent. If \( x_0 \) is the initial weight of the node and \( y \) is the weight of the parent node, and \( 0 \leq \alpha \leq 1 \) is the learning rate, then after the first weight adaptation of the node, its weight would be as in equation 1.

\[
x_1 = (1 - \alpha)x_0 + \alpha y
\]

If \( x_1, x_2, x_3, \ldots, x_n \) are the weights of the node after the weight has been adapted for \( 1, 2, 3, \ldots, n \) times successively and the initial weight of the node \( x_0 = 1 \) and the weight of the parent node \( y = 0 \), then the weight of the node after \( n \) weight adaptations would be as shown in equation 2.

\[
x_n = (1 - \alpha)^n
\]

If \( T \) is the threshold or the weight difference between the node and its parent which should be achieved for the node to be regarded as mature after \( n \) iterations, then equation (2) can be extended to equation 3 as follows,

\[
T = (1 - \alpha)^n
\]

C. Learning rate functions

The tree’s growth depends on the learning rate. All nodes in the tree can have the same learning rate, or they can have different learning rates depending on the depth of each node.

1) A constant learning rate function: If all nodes have a constant learning rate, the total number of sub sequences required for the subsequence corresponding to a node of depth \( m \) to be learnt by the tree, in the best case would be

\[
n = \log_{(1-\alpha)} T \]

Here the sub sequences would be presented in increasing order of length. But in the worst case, if they occur in decreasing order of length, it would be

\[
n_c = \log_{(1-\alpha)} T \ast m
\]

This would imply that longer sequences would be disadvantaged by using a constant learning rule. One could minimize this effect by presenting the input sequences randomly.

2) Increasing the learning rate: A better approach would be to use a variable learning rate function, where the learning rate increases with the node’s depth until it reaches a maximum value as given in equation 6 where \( \alpha_s \) is the start learning rate and the maximum learning rate of \( \alpha_f \) is reached at a depth of \( m_f \).

\[
\alpha_m = \begin{cases} 
\alpha_f \sin \theta & \text{for } m \leq m_f \\
\alpha_f & \text{for } m > m_f
\end{cases}
\]

where

Algorithm 1 The ASTree Algorithm

```
procedure INITIALIZE ASTREE(learning parameters)
    Create the root node
    Calculate the list of learning rates
end procedure

procedure TRAIN ASTREE(sequences S)
    for Each symbol c in each sequence of S do
        for The root & mature nodes of last symbol p do
            FindChild(p, prevSymbol, c)
        end for
        Prune non-mature nodes of the tree
    end for
end procedure

procedure FINDCHILD(PreviousNode p, PreviousSymbol sp, CurrentSymbol s)
    if sp is the last symbol of p then
        if p has no children then
            Append the symbol s to p
            Set p’s weight to 1 and set is as non–mature
            Adapt the weight of p
        end if
        if p has a child with first symbol as s then
            if the child is not mature then
                Adapt the weight of the child
            end if
            if p doesn’t have a child with first symbol s then
                Create node with first symbol as s & add to p
                Adapt the weight of the node
            end if
        end if
        if sp is an inner symbol of p then
            if the symbol after that is not s then
                SplitNode(p, prevSymbol, s)
            end if
            if the symbol after that is s then
                if p is not mature for s then
                    Adapt weight of p
                end if
            end if
        end if
    end if
end procedure

procedure SPLITNODE(Node n, PreviousSymbol sp, NewSymbol s)
    Create a newNode n1
    Set mature property of n1 equal to that of n
    Set weight of n1 equal to that of n
    Remove all symbols after sp from n and add to n1
    Remove all child nodes from n and add them to n1
    Add n1 as a child of n
    Set n’s mature property to true and weight to 0
    Create a newNode n2 with symbol s
    Add n2 as a child of n
end procedure
```
\[ \theta = c + \left( \frac{\pi}{2} - c \right) \frac{m - 1}{m_f - 1} \quad \text{and} \quad c = \arcsin \frac{\alpha_s}{\alpha_f} \]

Having a learning rate of \( \alpha_f \) for all nodes with depth larger than \( m_f \) could make the tree grow unconditionally. However, in certain situations it might be required to stop the growth of the tree after \( m_f \). This can be achieved by setting the learning rate for nodes with depth larger than \( m_f \) to zero. The learning rate equation for this scenario is given in (7).

\[ \alpha_m = \begin{cases} \alpha_f \sin \theta & \text{for } m \leq m_f \\ 0 & \text{for } m > m_f \end{cases} \quad (7) \]

3) Increasing and decreasing the learning rate: In certain situations, it might be a better to gradually decrease the learning rate after it reaches a maximum value, rather than abruptly making it zero. This would allow subsequences which are longer than \( m_f \) to be still captured if they are very frequent. The learning rate function could be defined in the form of a sine curve, such that it starts at \( \alpha_f \), then increases to \( \alpha_f \) and then gradually decreases to 0. The learning rate for a node of depth \( m \) is given in equation 8 where \( m_x \) is the longest sub sequence learned by the algorithm.

\[ \alpha_m = \begin{cases} \alpha_f \sin \theta & \text{for } m \leq m_x \\ 0 & \text{for } m > m_x \end{cases} \quad (8) \]

where

\[ m_x = 2(m_f - 1) + \frac{c(m_f - 1)}{\pi} + 1 \]

When using learning rate functions, during the initialization phase, it is a good practice to construct a lookup table for the learning rates of each level so that each node does not have to recalculate its learning rate but can acquire it through a simple cost efficient lookup. A graphical representation of the discussed learning rate functions are given in figure 9. These are some examples of learning rate functions. Others could be used according to application requirements.

D. Calculating the required number of sub sequences

\( n_m \) the maximum number of sub sequences required for a node of depth \( m \) to be extended can be calculated by (9)

\[ n_m = \log_{1-\alpha_m} T \quad (9) \]

Immediately after a node has reached a mature state, it would add the next node and adapt the new nodes weight. Therefore, from level 2 onwards the actual number of sub sequences needed can be calculated by (10)

\[ \overline{n_m} = \log_{1-\alpha_m} (T - 1), m > 1 \quad (10) \]

Therefore, the number of sub sequences \( n_t \) required for learning a sub sequence of length \( m \) can be obtained by (11)

\[ n_t = \left( \sum_{i=1}^{m} n_i \right) - (m - 1) \quad (11) \]

E. Calculating the learning rate for a sub sequence length

Based on the learning rate, the maximum number of sub sequences that should be present for a given learning rate and a given number of sub sequences can be calculated. Inversely the learning rate can be adjusted so that it captures sub sequences of the required length and frequency. The learning rate \( \alpha_m \) can be calculated as given in (12)

\[ \alpha_m = 1 - \sqrt[n_m]{T} \quad (12) \]

where \( T \) is the threshold and \( n_m \) is the number of sub sequences that should be present to learn the sequences of length \( m \), after the node of depth \( m \) has been added.

F. Training over multiple iterations

The same set of sequences can be input over multiple iterations into the algorithm. Increasing the number of iterations would have a similar effect as increasing the learning rate on the growth of the tree. The organization of the input space affecting the results could be avoided by randomly selecting the input and training for multiple iterations.

As discussed above, the algorithms’ learning parameters can be specified according to the length of the sequences that need to be captured. The ASTree algorithm is flexible such that the parameters can be defined in terms of the application.

IV. EXPERIMENTS AND RESULTS

We proposed the ASTree algorithm in this paper as a space efficient successor of the previously proposed ASTtrie algorithm. The ASTrie has been used in two diverse application areas. First in bioinformatics [14] which consists of long dense sequences and then in text clustering [16] which consists of short sparse sequences.
In this section we discuss the experiments carried out to demonstrate that the ASTree algorithm is more space efficient than the ASTrie algorithm. We measure the space in terms of the number of nodes grown by the algorithms. We ran the experiments on the same datasets that were previously used in the applications of the ASTrie and compared the number of nodes grown in both the ASTrie and the ASTree algorithms. We also compared the run time of the experiments. All experiments were conducted on a machine with an Intel(R) Core(TM)2 Duo 3.32GHZ processor and 3.46 GB of RAM under same runtime conditions. Each experiment was run three times and the mean of the runtime is presented.

### Table I

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of nodes</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASTree</td>
<td>ASTrie</td>
</tr>
<tr>
<td>20 Species (20 sequences)</td>
<td>26814</td>
<td>62814</td>
</tr>
<tr>
<td>34 Species (34 sequences)</td>
<td>44338</td>
<td>109729</td>
</tr>
<tr>
<td>HIV-SIV (70 sequences)</td>
<td>41770</td>
<td>137779</td>
</tr>
<tr>
<td>Reuters (7040 documents)</td>
<td>15099</td>
<td>37558</td>
</tr>
<tr>
<td>SMH News (18173 documents)</td>
<td>100051</td>
<td>230777</td>
</tr>
</tbody>
</table>

### A. Datasets

Experiments were conducted on three biology datasets and two text corpus. The biology datasets consisted of mtDNA sequences. These datasets were used previously in [14] and they have been extensively used in bioinformatics for phylogeny analysis [25], [26]. DNA sequences consist of 4 nucleotides, Guanine (G), Adenine (A), Thymine (T), and Cytosine (C) that combine together to form long sequences [3]. The first and the second datasets are the 20 and 34 species datasets consisting of sequences of length ranging from 16,299 - 17,245 nucleotides. The third dataset consisted of 70 HIV-SIV sequences of length 8555 - 11,443.

The two text corpus were previously used in [16]. The first was the Sydney Morning Herald (SMH) news archive consisting of 18173 news articles published in Sydney Morning Herald in 2009 in the 4 main news channels National, Sport, World and Entertainment. The second was the Reuters 21578 distribution 1.0 dataset consisting of 7040 documents in its 4 main categories, Acquisition, Earnings, Interest and Trade. Text is sparser when compared to sequences found in biology, since they consist of a large vocabulary. In addition, repetitions in text are found in the form of phrases. It has been identified that phrases with length larger than three would reduce the performance of clustering [27]. Therefore text phrases up to a length of three at maximum learning rate was chosen in the study carried out in [16]. Hence, we looked at extracting shorter sequences from the text datasets.

### B. Comparison of space and time

Experiments were run for a constant learning rate function as well as a variable learning rate function. A constant learning rate of 0.5 was used for all datasets. We used a variable learning rate function in which the learning rate was gradually increased and then decreased as in equation 8. Parameters were chosen to be identical to the experiments carried out for the ASTrie in its previous applications [14], [16]. For the biology datasets, which consisted of longer sequences, a start learning rate of 0.1, a maximum learning rate of 0.9 and a length of 30 at the maximum learning rate was used. For the text datasets, which consisted of shorter phrases, we used a start learning rate of 0.1, a maximum learning rate of 0.3 and a length of 3 at the maximum learning rate. The threshold was set to 0.1 for all experiments.

We recorded the number of nodes grown for the ASTree and the ASTrie as well as the time taken for the experiments. The results for a constant learning rate are given in table I and for a variable learning rate are given in table II. The percentage decrease of nodes in the ASTree when compared to the ASTrie is given in the last sub column under the column number of nodes in both tables. It can be seen that there is a node reduction of over 50% in all cases in the ASTree when compared to the ASTrie. This ranges from a minimum reduction of 50.2% for the SMH news dataset and a maximum reduction of 72% for the HIV-SIV dataset as shown in table II. For all datasets, the percentage of reduction increases with the increase in the number of sequences. This is due to the increase in the number of distinct sub sequences, when the number of sequences increases. The reduction in nodes is much larger in the biology datasets when compared to the text datasets. The reason for this is that biology datasets consist of longer repetitive patterns when compared to text. Since nodes are compressed with respect to their depth, it results in a higher reduction in nodes for biology datasets.

It can be seen that there is an increase in the time taken for calculation. This is due to the additional complexities of the ASTree algorithm when compared to the simpler ASTrie algorithm. However, the time increase happens in terms of seconds and the percentage increase in time is less than half the percentage decrease of the number of nodes. This can be regarded as an acceptable increase when compared to the reduction of nodes. Large space utilization is the main problem associated with suffix trees and tries. The reduction in size of the suffix tree results in it chosen over the suffix trie [17]. Due to the same reason the ASTree would be preferred over the ASTrie for real world applications.
V. DISCUSSION

In this paper we presented the ASTree algorithm for learning frequent variable length sub sequences from a given set of sequences. The ASTree algorithm is the successor of the ASTrie algorithm presented previously. Similar to the ASTrie the ASTree algorithm is based on the Hebbian learning principle, a biologically inspired learning rule. The ASTree algorithm incorporates Hebbian learning into suffix trie data structures. It has the advantage that it is space efficient when compared to the ASTrie. Experiments demonstrate that over 50% of the space utilized by the ASTrie can be reduced by using the ASTree.

The next step in this series of work is to enable the ASTree for learning continuous sequences. The flexibility of the Hebbian learning rule makes this a feasible future enhancement. Data is the real world is rarely discrete. Enabling the ASTree to work with continuous sequences such as speech or time series data would be highly beneficial. Therefore, the full advantage of incorporating learning into suffix data structures would arise with continuous sequences because it would create a more flexible structure which would have the adaptability and the flexibility to learn and capture sequential patterns from dynamic data. In addition, we have used a simple algorithmic approach for constructing the ASTree, mainly done for demonstrating how the Hebb rule can be incorporated into suffix data structures. It would be interesting to see how more complex but space and time efficient algorithms used for constructing suffix trees could be adapted for constructing the ASTree. With all these future improvements in place the ASTree would open up sequential information processing pathways in diverse application areas.

ACKNOWLEDGMENTS

The authors would like to thank Reuters and Sydney Morning Herald for their datasets which are used in this paper.

REFERENCES