Motif Discovery using optimized Suffix Tries

Sergio Prado

Promoter: Prof. dr. ir. Jan Fostier

Supervisor: ir. Dieter De Witte

Faculty of Engineering and Architecture

Department of Information Technology – IBCN (Internet Based Communication Networks and Services)

Academic Year 2011-2012

Acknowledgments

I would like to thank Dieter De Witte for all the help and guidance provided and for his unlimited patience during all these months.

I would also like to thank the IBCN and UGent staff for giving me this great opportunity.

I must thank Juliane for her help in language issues and my brother Javier for being always ready to answer biology questions in a "kindly" way.

I would not like to forget to thank Francesc Manel Martínez Villar for pushing me to start this new adventure.

And also thanks to all those new people I met in Gent and shared with me this great year.

0. Index

1. Table of Symbols

Number of Sequences	N
Sequence Length	n
Motif Alphabet	$\boldsymbol{\Sigma}$
Alphabet size	$\lvert\boldsymbol{\Sigma}\rvert$
Pattern	P
Pattern length	$\lvert P\rvert$
Maximum pattern length	k
Degeneracy grade	S
Quorum	q

2. Abstract

3. What is Motif Discovery?

Proteins take part in all biochemical reactions in living beings. The information to build them is stored in a cell's DNA (deoxyribonucleic acid), and the information is retrieved in a 2-step process (Figure 3.1).

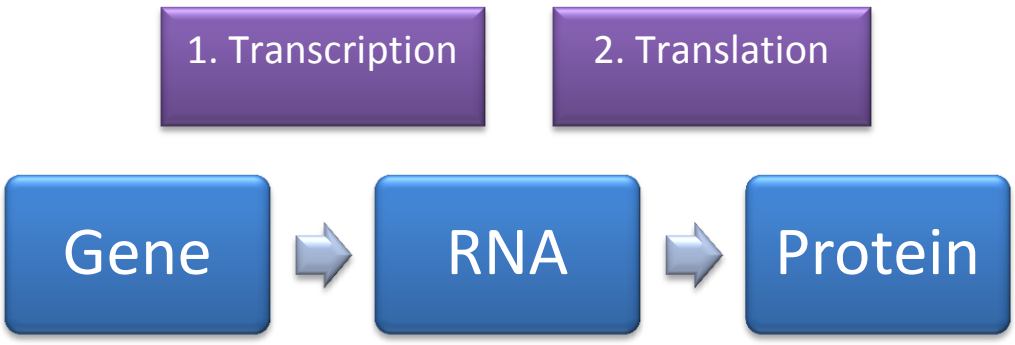

Figure 3-1 - 2-Step process to obtain protein from a gene

The first step, transcription, occurs in the cell's nucleus. The cell's machinery copies the gene sequence into messenger RNA (ribonucleic acid), a molecule that is similar to DNA. This messenger RNA travels from the nucleus to the cytoplasm where the translation starts. During translation, the ribosome, a protein-making machinery, reads the messenger RNA sequence and translates it into the amino acid sequence of the protein [1].

We are looking for the binding sites for transcription factors, which have as a function promoting or blocking the recruitment of RNA polymerase. This is the molecule that reads DNA and creates messenger RNA. These binding sites are short DNA segments which are important for genome regulation and are usually degenerate. Degeneracy is the redundancy of the genetic code, and can be defined as the ability of elements that are structurally different to perform the same function [2]. This implies that small variations of a certain binding site are allowed without destroying its biological function.

Identifying binding sites is a crucial and important task in understanding the mechanisms that regulate gene expression. [3]

Detecting the binding sites requires a model to represent them, which we call a motif. We use a word-based model based on the IUPAC alphabet [4] (Figure 3-2).

Code	Description
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
R	Purine (A or G)
Y	Pyrimidine (C, T, or U)
M	C or A
K	T, or G
W	T, or A
S	C or G
B	C, T or G (not A)
D	A, T or G (not C)
H	A, T or C (not G)
V	A, C, or G (not T)
N	Any base (A, C, G, T)

Figure 3-2 - IUPAC Alphabet Table

Motif discovery is a challenging problem from a computational point of view [5] [6]. Binding sites are better conserved in DNA because they have a biological function and are therefore under selective pressure. Motif discovery algorithms can help us detect them.

To tackle our problem we design and implement an index structure and a motif discovery algorithm. In this thesis we will investigate memory and performance optimizations.

An introduction to motif discovery can be found in Pevzner[5]. Several approaches have been tested for example exhaustive searches using profile matrices, casting the

problem into a median string problem, using search tries… but all this results in disappointing runtime. One approach that can give us better results is based on a greedy approach which has $\mathrm{O}(\mathrm{N} \cdot \mathrm{n}^2)$ but this approach cannot guarantee that the optimal motif will be found.

For a recent survey on motif discovery we refer to Das [7]. They classify the motif finding algorithms in three groups, based on the type of DNA sequence information employed:

1. Those that use promoter sequences from genes involved in the same process from a single genome

This category includes the word-based methods that rely on exhaustive enumeration, and which can be implemented using structures like a Suffix Tree [8] or a Suffix Trie. There are also the probabilistic algorithms, for example CONSENSUS [9]. The probabilistic algorithms consist of Gibbs sampling methods and Expectation Maximization Strategies. They use methods to optimize a score of given information content.

2. Those that use orthologous sequences from multiple species

The standard method used for phylogenetic footprinting, where the functional elements are expected to be conserved, is to construct a global multiple alignment of the orthologous sequences and then identify conserved regions in the alignment. An example of a popular alignment tool is CLUSTAL W [10] or DIALIGN [11].

3. Those that use promoter sequences of coregulated genes as well as phylogenetic footprinting

These algorithms integrate the two important aspects of a motif's significance into one algorithm.

Our search strategy is based on the first group of algorithms, but we are using it for the second type of data in the classification. We will use Suffix Tries to discover these repeated regions in the DNA sequences. This leads to an exhaustive motif discovery algorithm where a lot of motifs will be tested, but we will use a branch & bound technique to limit the candidate space.

As a representation we choose a Suffix Trie. In previous works a Suffix Tree was used [9]. We want to investigate the performance of this structure. The Suffix Trie has memory complexity $O(n^2N^2)$. We can reduce it to $O(knN)$ complexity which is linear for our type of problem. Since every path in the trie is unique, we can find all occurrences of a pattern in $O(S|P||\boldsymbol{\Sigma}|)$ time. This cost is obtained as a result of the product of three different factors. The first value S is the degeneracy of the motif,

which is the number of exact binding sites matching with it. Similar sites can perform similar functions as we explained before. For example, if we introduce two mutations in the sequence *ANGTNC* where *N* has been used to denote these mutations, the sequence has degeneracy 16, which is the product of the degeneracy of its characters (Figure 3-3).

Figure 3-3 – Possible motifs in a sequence with 2 localized mutations in a 4-character alphabet, degeneracy=16

This implies that *ANGTNC* corresponds to 16 unique paths in the trie. One path has length $|P|$, so requires $|P|$ character comparisons. To find a pattern *P* we will follow the unique path that spells the motif.

The last value $|\boldsymbol{\Sigma}|$ indicates that we might have to do $|\boldsymbol{\Sigma}|$ horizontal steps at every node, because for a single node we have to compare in the worst case, the current character with all the characters in the alphabet.

This is an essential feature for motif discovery when we look for *k*-words because our search in the trie is really fast as compared to a naïve scanning which requires to go through the complete sequence. In our case, when we use suffixes the cost in time will be $O(kNn + \#P \cdot S|\boldsymbol{\Sigma}||P|)$ to find *#P* patterns. The main advantage is that looking for many patterns is really fast while we only need to build the trie once.

An important optimization of the Trie is that we can stop the construction of the trie at a certain depth, so we can handle large data sets without the obligation of creating a complete Suffix Trie, which is the reason we obtain $O(knN)$ as a construction and a memory cost. If we stop the construction of the Trie at *depth=k*, we will handle all the different substrings with *length≤k*. The repeated strings share the same path which will help us to save more memory. This is the reason we only require linear amount of memory.

In comparison with sliding window approaches we can easily see that if we have to look for many patterns we will have the advantage that we only need to build the trie once and we will realize faster searches. The sliding window approach, does not require us to build a structure, but the lookups are slower since we need to iterate through all the data which results in a bad performance for this kind of task.

The implementation can be subdivided in two phases: the first one is the design of the data structure, and a second one is the design of motif discovery algorithm.

The main issues we have to deal with concern the memory and performance. We will try to design a structure using the minimum memory possible avoiding the use of fields and pointers. This will help us to obtain a data structure with a good general performance and the ability to solve problems with a huge amount of data.

First we are going to introduce some concepts. A suffix trie contains all suffixes of the sequences we are indexing. Those suffixes are substrings which start between the position 1 and the end of the sequence, and always end with the last character of the sequence. In the trie we store parts of those suffixes which are all the substrings with length between 1 and k, where k is the deepest level in the trie. So after the construction of the trie, our structure will handle all the substrings with maximum length k that appear in the sequences.

The current implementation of the suffix trie at IBCN requires 44 bytes per character while we are able to optimize this to 16 bytes. This will be explained in the next chapter.

4. Efficient Memory Structure

In this part we will discuss the data structure used for motif discovery. We try to use the minimum memory requirements for our data structure and this will determine the performance of our algorithm both in memory and time.

4.1. Suffix Trie Representation

In our trie each node is connected with its left child and its right sibling. Each node can have 1 to $|\Sigma|+1$ children (one of the children can be the sentinel character indicating the end of the suffix). With this design for a node, we only need one pointer to the leftmost child instead of having $|\Sigma|$-pointers per node. It adds a factor Σ to the pattern matching, but since we need to iterate through the complete trie this does not cause loss of efficiency. We will explain later how to get rid of the sibling pointer.

The node size does not depend on the alphabet size as is the case with the usual structure. This saves a factor Σ for the memory. A suffix can be represented using two Integers, the first one to indicate at which sequence it belongs and the second is the position into the sequence where it starts.

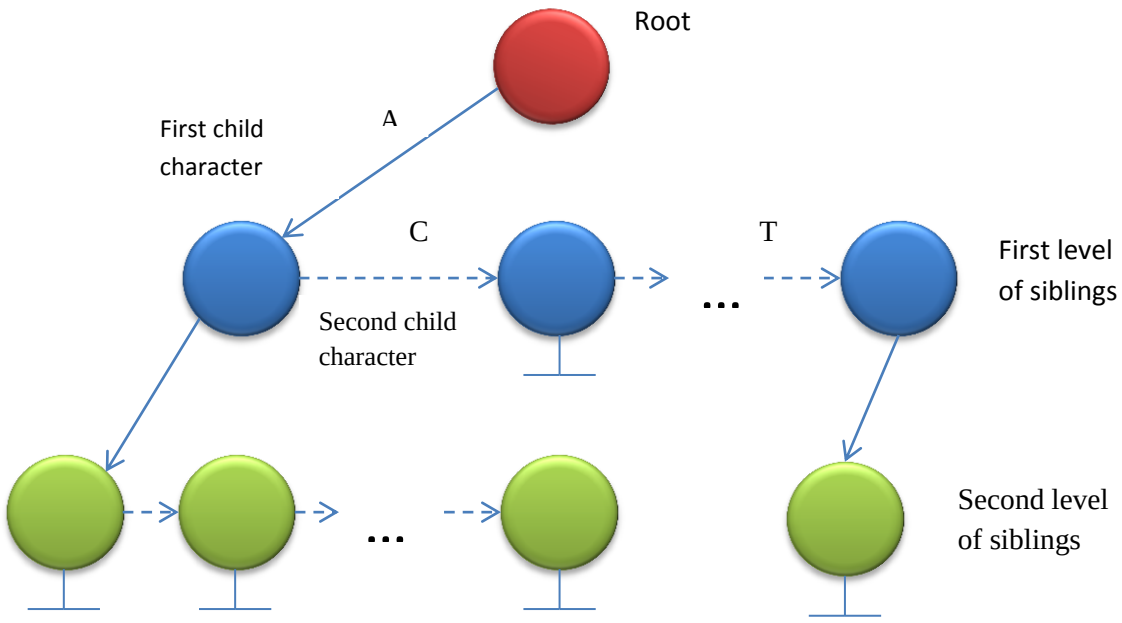

Figure 4-1 - Suffix Trie structure

We can avoid pointers between siblings by allocating them next to each other in the node vector, therefore we only need a boolean to indicate the rightmost sibling.

In each node we have as information the character and the links to the next sibling and to the first child as we can see in figure 4-1.

The red node is called root and its children are all the different characters that appear in the sequences.

4.2. Generalized Suffix Trie

When working with multiple sequences we use a Generalized Suffix Trie which contains additional sequence information in the internal nodes and leafs. The Generalized Suffix Trie (GST) can be built in $O(kNn)$ time and space [6], and it takes again $O(S \cdot |\Sigma| \cdot |P|)$ to find all occurrences of a string of length $|P|$ in a GST.

4.2.1. Trie as a vector of nodes

The trie is stored in the memory as a vector of nodes. In each position all the information related to a single node is stored (which is represented in Figure 4-2): the character of the incoming edge, the index of the leftmost child and a boolean indicating if the actual node is the rightmost sibling. With those indexes we can iterate through the trie.

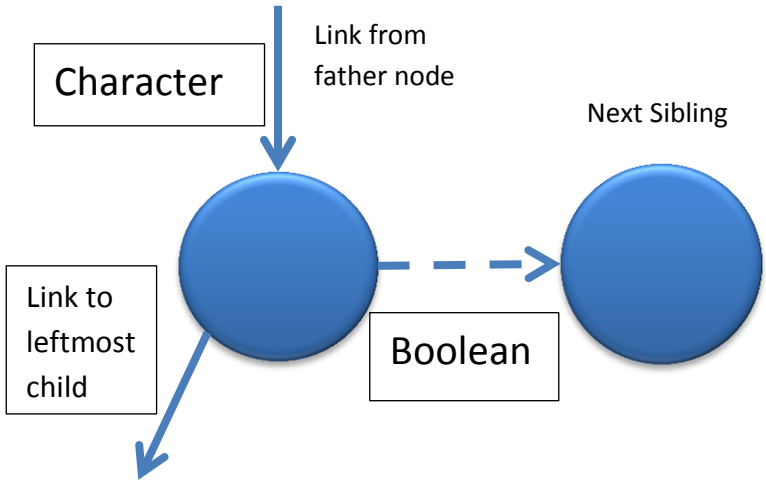

Figure 4-2 - Representation of a single node and its attributes

We are able to spend only a boolean per child which has size 1, instead of one link from the father to each of its children, because since we are using a vector, which elements occupy contiguous memory location, we can move easily from the leftmost child to the next siblings if they are allocated together. To move from one child to the next one, we will increase the current index by 1 if the *boolean* indicates that the actual child is not the last sibling. And we will repeat this until we reach the last child which its *boolean* indicates is the last one.

This design of the trie implies that each node occupies:

$$\boldsymbol{sizeof(node)} = sizeof(char) + sizeof(int) + sizeof(bool) = 1 + 4 + 1 = \boldsymbol{6\ bytes}$$

In Figure 4-3 and Figure 4-4 we can observe the representation of the sequence "CATA" as a suffix trie and as a vector. In the representation of suffix trie, each link between father and first sibling is represented with a straight arrow, and the dashed lines represent the virtual link between siblings.

Each node appears with the information stored in each position of the vector (character, child index and last sibling flag). And for those nodes that are leafs, the box with the string corresponds to the suffix at that point, this is not actually stored in the trie since corresponds to the concatenation of characters we meet when moving from root to leaf.

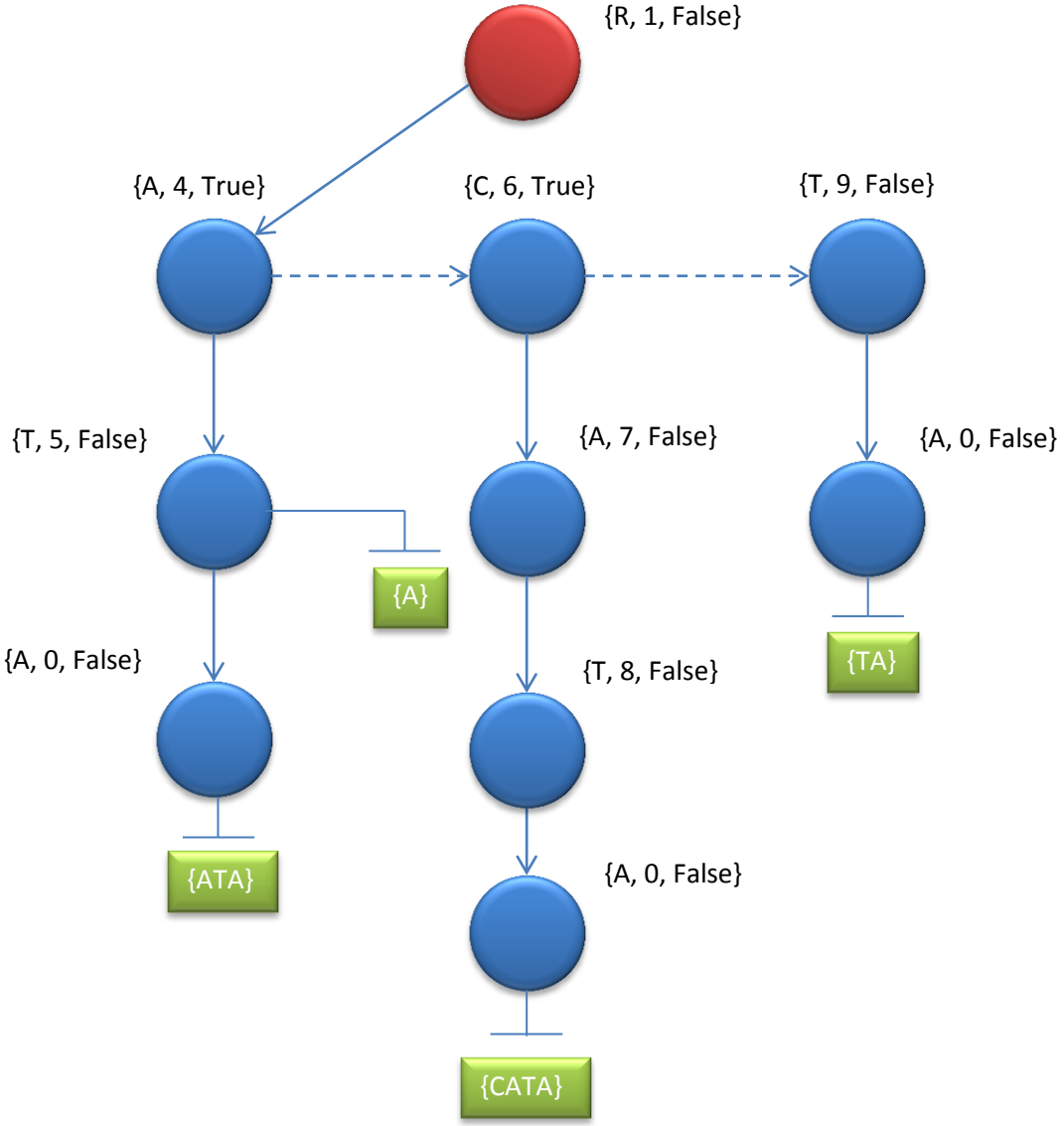

Figure 4-3 - Suffix Trie of the sequence "CATA"

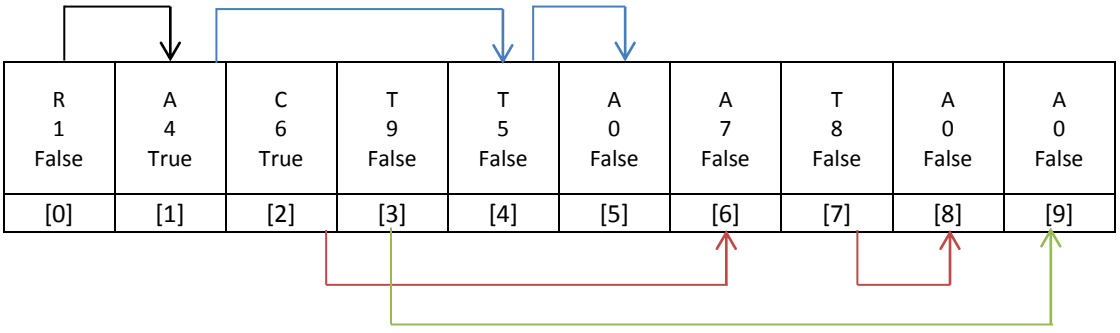

Figure 4-4 - Vector containing the sequence "CATA"

4.2.2. Additional Structures

Complementary to the node vector we use two hashmaps containing the sequence and leaf information. This additional structures will help us to perform faster queries. This information is defined as:

Suffix Information: the sequence ID and the index of the suffix

Sequence Information: the set of sequences IDs associated with a node.

Those structures will allow us to perform two queries:

1. Where does a pattern occur?
   We will look for in the trie to find the suffixes below a pattern in the trie and compare them to a pattern.
2. In how many sequences does the pattern appear?
   This information will help us to discard the non-promising motifs and to process only the interesting ones. This can be calculated as the size of the *Sequence Information* set in a specific node.

4.2.2.1. Leaf Information structure

The first map contains the leaf information, where the key is the index of the leaf node in the node vector and the item is a vector which contains the suffixes. A *Suffix* contains two fields, the first one is the *sequence ID* and it is represented by a 32 bit integer, and the second one is the *suffix index* which is represented by a 16 bit integer. So the structure needs 6 bytes per character for each node. We can change the 32 bit integer field that represents the *sequence ID* into a 16 bit integer, which implies that it is not possible to allow more than $2^{16} - 1\ sequence \approx 64.000\ sequences$, but we can save some space here. This configuration can easily be modified.

We are using a vector for the *Suffix Information*. This is necessary because a leaf could contain multiple suffixes, and since we are cutting the trie at a certain depth, we can have more than one suffix in the same leaf and we have to store them.

We can conclude then that we need for each leaf **16 bytes**: 6 bytes per node in the vector, and 10 bytes for the *Suffix Information* in the leafmap:

- 1 integer for the index of the leaf node
- 1 integer for the sequence ID
- 1 short for the suffix index

$$\boldsymbol{sizeper(leaf)} = sizeof(node) + sizeof(leafInfo) = 6 + 10 = \mathbf{16\ bytes}$$

4.2.2.2. Sequence Information structure

The second map contains the information that will help us to find out in which sequences a pattern occurs. This map has an integer representing the index in the node vector as the key, and the item is the Sequence Information, being the set of sequence IDs. Thus we easily know if a pattern appears in a sequence just by checking if the sequence ID appears in the set. The construction of this structure will be done by using a bottom-up approach once the trie is already built which will be explained in section 4.4.

We will only add this information in the splitting nodes, which have two or more children. This way the structure remains linear in memory not affecting the memory complexity of the structure, since there are only *O(nN)* splitting nodes, the same as in Suffix Tree.

The memory required per each element on the Sequence Information is:

$$\boldsymbol{sizeof}(\boldsymbol{SeqInfo}) =$$

$$sizeof(int) + sizeof(set) \cdot \#elements = 4\,bytes + 20bytes \cdot \#elements$$

4.3. Suffix Trie Construction

The method which is used to build the Suffix Trie is based in the Write Only, Top Down (WOTD) algorithm [12], and an example of a sequence diagram is shown in Figure 4-5. It is a Depth First algorithm and keeps extending a path until a leaf is reached instead of build it suffix by suffix. We use queues to push all the suffixes that start with the same characters, and we continue expanding nodes in a recursive fashion.

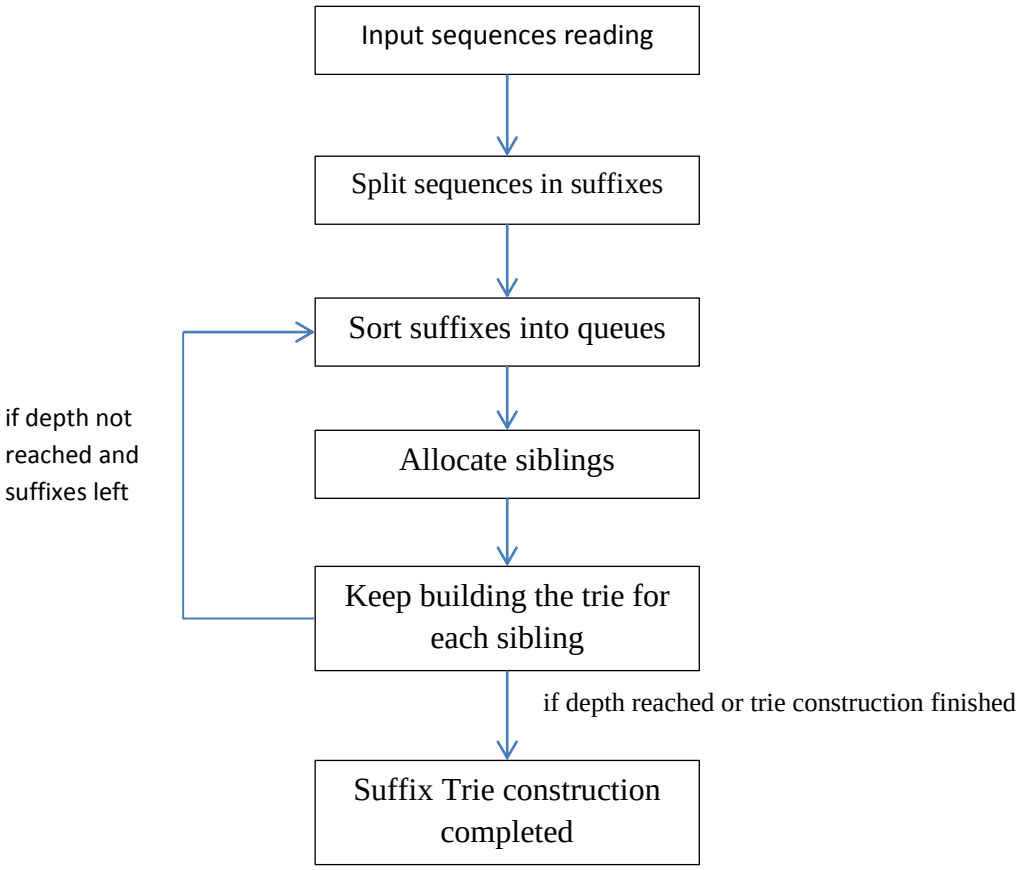

Figure 4-5 - Suffix Trie construction Sequence Diagram

Our program is able to read a set of FASTA format sequences with the use of any alphabet, e.g.:

$$> Name\ sequence0 - Description$$
$$CATA$$
$$> Name\ sequence1 - Description$$
$$TATA$$
$$\ldots$$

After reading the file, an array of sequences is sent to the constructor. At the first phase the constructor creates the root node and adds a sentinel character (Figure 4-6) to all the sequences and creates the different suffixes of the sequences (Figure 4-7). The sentinel character ('$') is added to be able to know when the end of the suffix is reached.

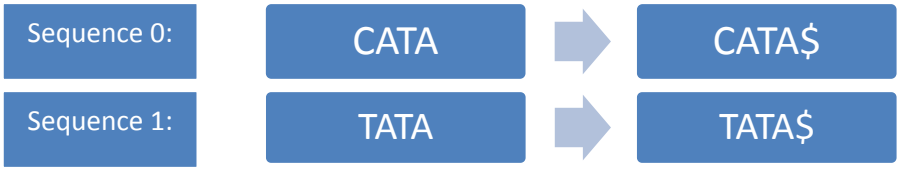

Figure 4-6 - Adding sentinel character to sequences

During the process of sorting the different suffixes we are using an auxiliary structure called *VectorQueue* to sort them into different queues. Each position of the *VectorQueue* contains one character and holds the suffixes which share this character in a queue. For each new character a new position in *VectorQueue* is created with a new queue and subsequent suffixes with the same character are added in the already existent queue.

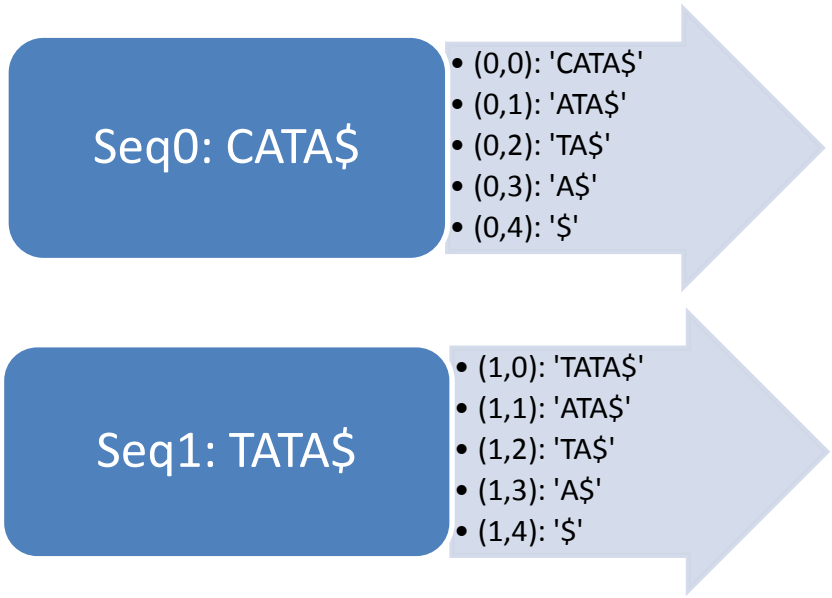

Figure 4-7 - Creating suffixes from sequences

Once we have sorted the suffixes into different queues (Figure 4-8), the nodes are created and added to the vector of nodes. Since we created a queue for every character appearing in the sequences we only need to iterate over all the queues and to create one node for each existing queue.

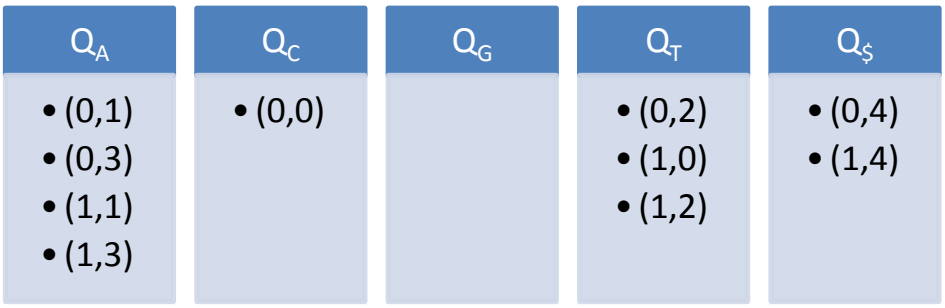

Figure 4-8 - Suffixes sorted into different queues

After this phase we have a vector which contains the root and the first level of siblings being allocated next to each other and the last one with the flag indicating that there are no more siblings (Figure 4-9). At this point the recursive function is called for each child of the root giving as arguments the queue, the *Suffix Information* and the *string depth* initialized at 0.

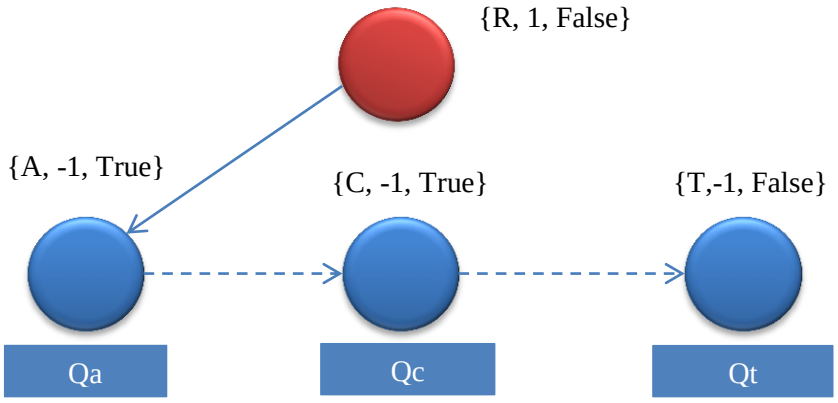

Figure 4-9 - Suffix Trie after first iteration.

The use of the variable *string depth* is required since in each recursive call since we go one level down and we move one position forward in the sequence. Thanks to this we can manage to figure out which position of the sequence we are currently processing.

Another parameter in the Suffix Trie constructor is the *cutoff depth* we want our Suffix Trie to reach. This value has an important impact on our algorithm performance. It is used as one of the two termination conditions in our recursive construction algorithm:

1. If we reached the maximum depth *k*
2. If we meet the sentinel character

The type of recursive function implemented is depth first recursion, calling itself as many times as different characters exists. In each call the depth is increasing reducing the problem size.

The algorithm sorts the suffixes in different queues of VectorQueue using the function *sortQueues*. For each queue a new node is created and added in the vector. Afterwards we expand the next branch and we keep the index of the first sibling.

When all the siblings are allocated we can call the recursive function for every one of them. We do this afterwards because otherwise the siblings should be allocated next to each other.

For each queue we call the recursive function increasing the value of *string depth,* and after that we update the leaf information of the sibling which we have expanded since the entire suffix subtrie is already built.

The last step is to return the index of the first sibling which we stored before, which we will use as the access point from the father node to the rest of subtrie.

The auxiliary function *sortQueues* receives the current *step* value, the set of sequences, the queue associated to the node which we are expanding and a structure were store the new queues resulting of splitting the suffixes in the current node as arguments. With the information stored in the queue we can determine the suffix index, and add this value to the current *step* and then we have the actual character to be processed. As we did before, we create a new queue or we store it in an existing one now.

When all the recursive calls finish, the root node receives the index of the first child, and the trie is completely built as it is possible to observe for the sequences “CATA” and “TATA” in the Figure 4-10.

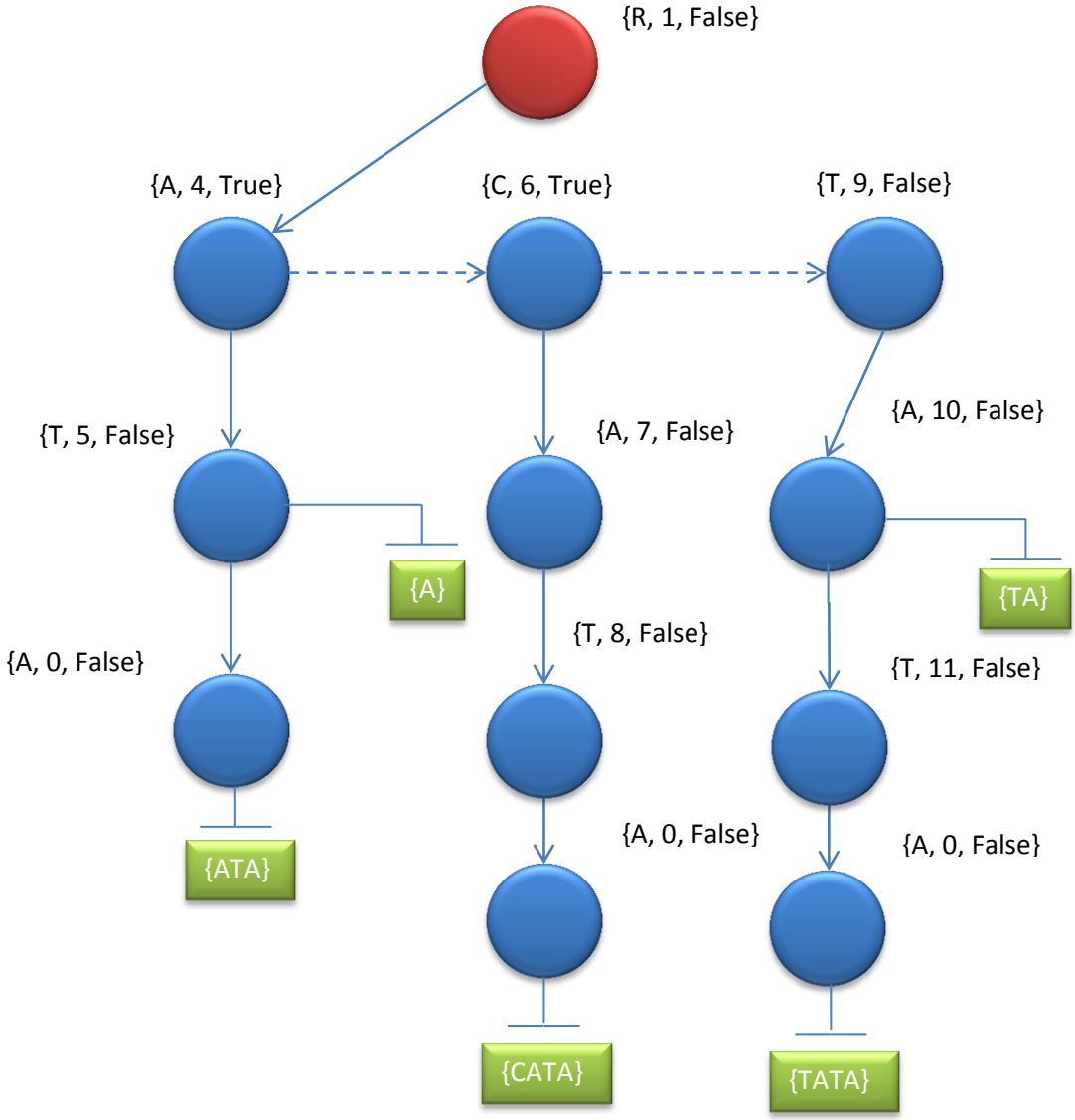

Figure 4-10 - Resultant Suffix Trie for the sequences "CATA" and "TATA".

4.3.1. Leaf Information structure

To know which characters correspond to the suffixes we need the suffix index and the *string depth*. We store this information when during the suffix trie construction, we reach a leaf. When the leaf is reached we add this information to the Leaf Information structure.

4.3.2. Sequence Indexes Structure

After completing the construction of the Suffix Trie we use a bottom-up strategy to store the information related to the sequences indexes for the splitting nodes (being the nodes with two or more children). The idea behind the function is simple: using recursive calls, starting at the root node, we reach a leaf where we add the different sequence IDs that we pick up from the Leaf Information Structure and send it to the father node. The node, who is receiving this information, takes the union of the sets in its children, stores it, and passes it to its father (Figure 4-11).

Since we only visit every splitting node once this function is also linear in the sequence length.

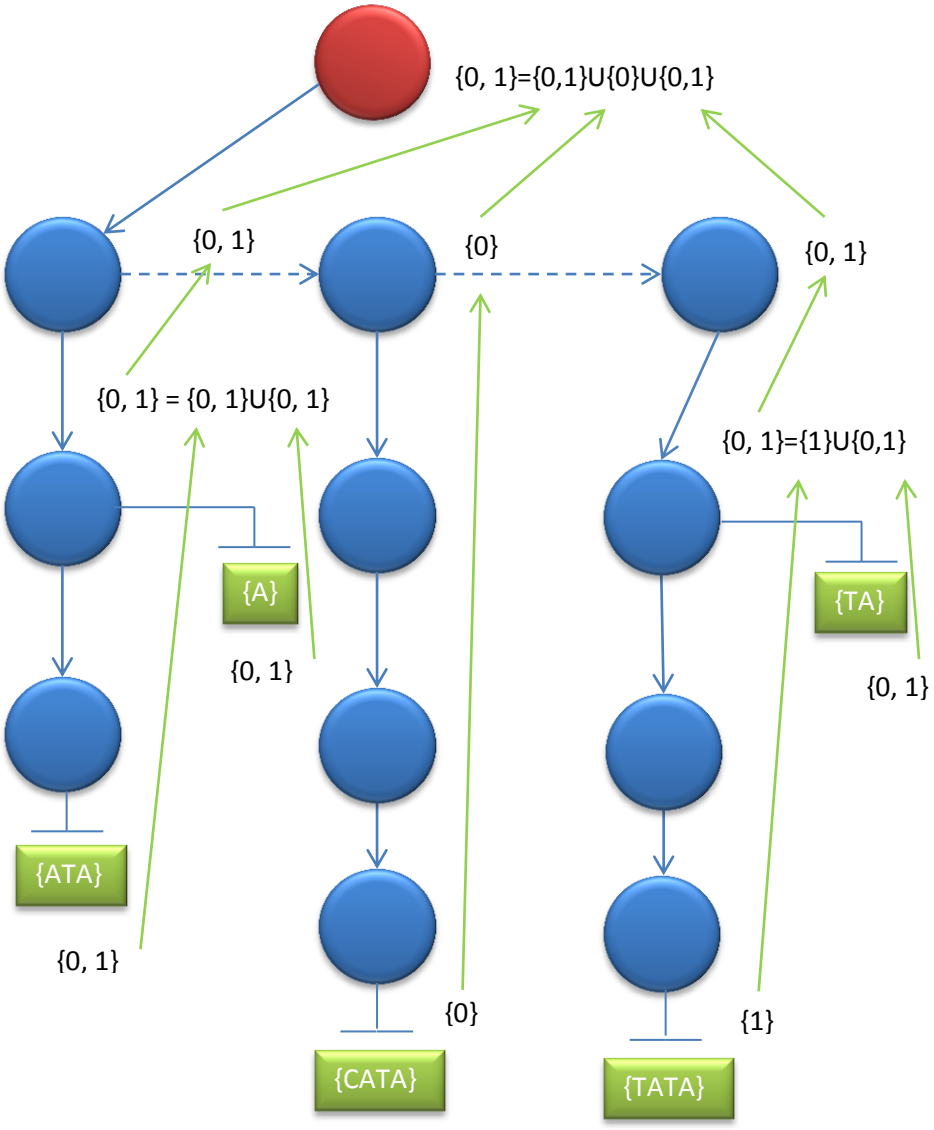

Figure 4-11 - Sequence Indexes structure construction: Sequence 0-> CATA, Sequence 1-> TATA

4.4. Validation of the construction algorithm

The purpose of this section is to check whether the recursive function is correct.

We will consider the next alphabet: $\Sigma = \{ A, C, G, T\}, | \Sigma | = 4$, although our program can manage different alphabets. We consider the value of the *cutoff depth* as ∞ for the worst case, which is when we need to expand all the nodes to reach the leaves.

In first stages, we drew by hand a Suffix Trie and also the additional structures. After the simulation we compared the structure built by our algorithm with the structure we drew before.

For bigger examples we devised a unit test.

One way was using the results from a Suffix Tree approach, which construction algorithm was truly reliable. The fact is we can establish a relation between the number of nodes from a Suffix Tree and from a Suffix Trie. For example the number of leafs is the same in both structures. The strategy was to count the number of nodes with a fixed number of children and then comparing both results to check it. In both structures the results should be the same for all nodes with 2 or more children.

The same strategy was used to check the total number of nodes. In the Suffix Tree, there are no nodes with only one child, so we compared its total number of nodes with the total number of nodes minus the number of nodes with one child in our structure.

Once the main structure is checked, is time for the additional structures.

The Leaf Information structure size is easily verifiable since we only need to compare it with the number of nodes with any child, because for each child we create a new entry in the Leaf Information structure.

In the Sequence Indexes structure we create an entry for each leaf or splitting node, so the way to check it is comparing the number of entries in this structure with the total number of nodes in the trie minus the nodes with only one child to avoid then the non-splitting nodes. If both numbers are the same, then we are storing all the splitting nodes and leafs in the structure.

We also devised a unit test using a sliding window approach. With the sliding window we can easily get all the different substrings of length k in the sequences so we can have the total number of substrings. Then we need to compare this with the number of nodes in our trie.

Using that we only need to compare the number of *k-mers* stored in our trie iterating through it reaching all the leafs and the set sizes obtained with sliding window as it is showed in Figure 4-12.

k	Number of nodes at depth k in the trie	Number of words using sliding window approach
2	16	16
3	64	64
4	256	256
5	1024	1024
6	4096	4096
7	16384	16384
8	65536	65536
9	256438	256438
10	644453	644453
11	889787	889787
12	970860	970860
13	992483	992483
14	997921	997921
15	999365	999365
16	999740	999740

Figure 4-12 - Example of validation of Suffix Trie construction using sliding window approach for 10 sequences of length 100.000 by comparison of the number of substrings depending on the size of the substrings

5. Motif Discovery in an extended alphabet

In this chapter we will explain and discuss our motif discovery algorithm.

Once the Suffix Trie and the additional structures are built, we are ready to run the discovery algorithm. In motif discovery the suffix trie is traversed in a Depth-First (DFS) way. The candidate motifs are those that occur in two or more sequences. Their path and their lengths are between a *k_min* and *k_max* specified.

We will subdivide the motif discovery in two categories: exact and degenerate. The difference is that the second one also searches for degenerate motifs in our sequences.

We will use a class Alphabet, taking into account all properties of the IUPAC symbols. This way the algorithm is decoupled from the alphabet type. We use several parameters to select our motifs. We set the motif lengths (*k-min* and *k-max*) allowed, the minimum number of sequences on which a pattern appears (quorum constraint) and the maximum degeneracy level (S). Usually we will set the minimum and maximum lengths of our motifs between 5 and 16, and the degeneracy allowed at 16, but these values can be easily changed. For exact motif discovery the degeneracy value is 1.

5.1. Exact Motif Discovery Algorithm

The exact motif discovery algorithm searches for all the sites that appear in exactly two or more sequences with a length between *k-min* and *k-max*. The algorithm is based on DFS and uses two terminations criteria: a branch and bound condition and the maximum motif length. The branch and bound condition can be easily formulated as follows:

> *When a motif α occurs in less than q sequences, all the possible motifs we can build adding characters from our alphabet to the current motif α(Σ)* also will occur in less than q sequences.*

In our trie it implies that when we are visiting a node which occurs in *q* or less sequences, we do not have to expand the path from this node.

When we are visiting a node during the execution of the algorithm we use the class Alphabet to try to move from the actual node to the next node. As an example, we choose the sequences: CATA and TATA. We look for motifs that appear in both sequences with length between 2 and 3, the first iteration through the trie using Σ={A,C,G,T} will allow us to move to nodes 1, 2 and 3, using the characters A, C and T respectively.

For each different character we will find a unique path to the next, so to keep iterating we will concatenate this character to the actual motif and increasing the length of the actual motif.

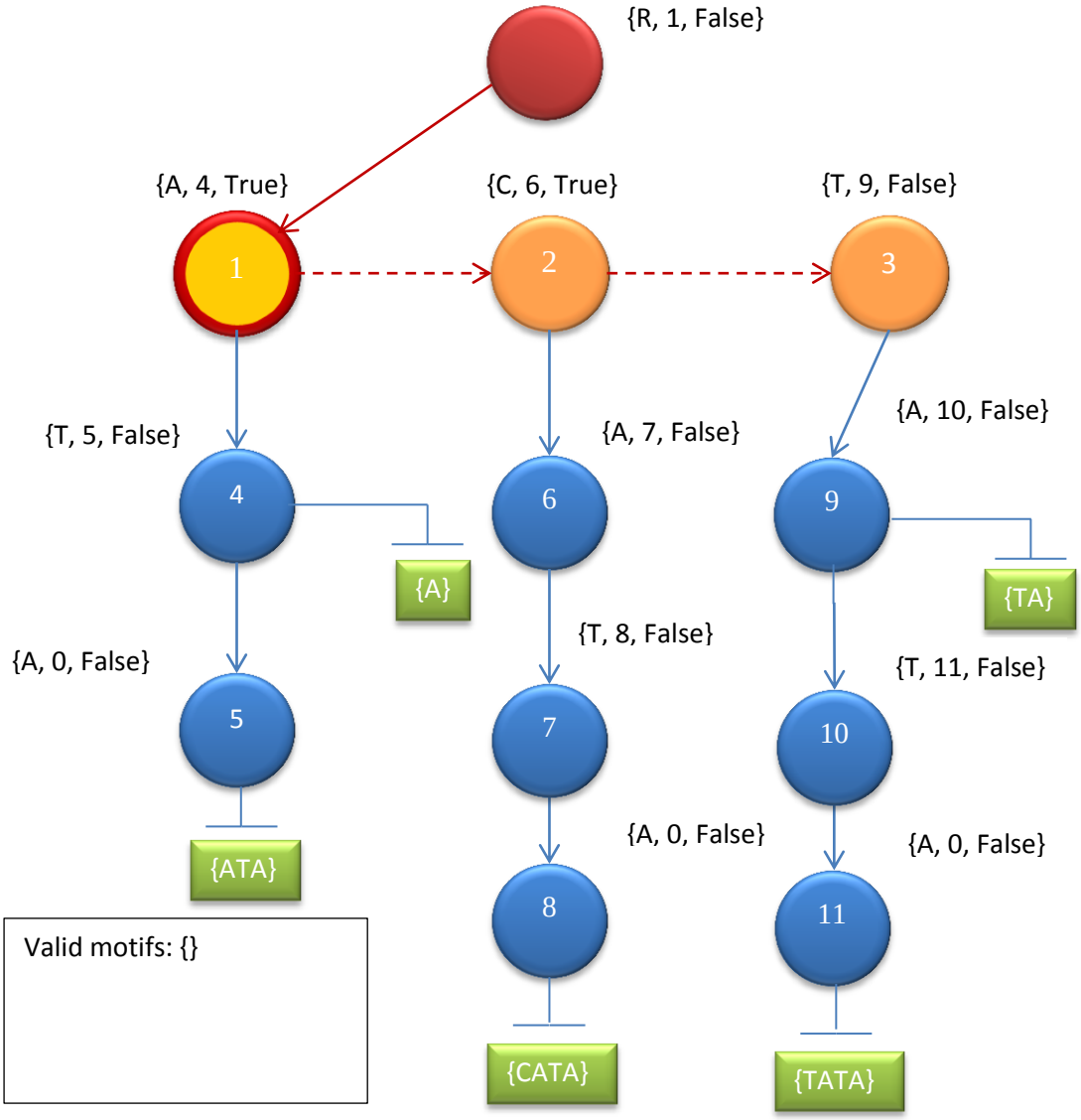

Figure 5-1 - In orange, nodes to expand in first iteration using characters: A, C and T from root node. In yellow with red ring, actual node being visited.

We store in a vector those nodes that we have to expand (1, 2 and 3) and using DFS we start iterating through the first path processing node 4. From this node we will try to

add a new character to the actual motif “AT”, where the only one possible path is adding an “A”: “ATA” (at node 5).

From node 5, in the next iteration the algorithm cannot iterate using any character from the alphabet, so the iteration through this path is stopped and it is time to check the current motif. The motif “ATA” appears in both sequences 0 and 1, and length is 3. It means that the algorithm found a correct motif, which is added to a list before returning the execution to the father node.

In the father it is not possible to iterate through any other character, so the current motif “AT” is checked resulting also in a valid motif.

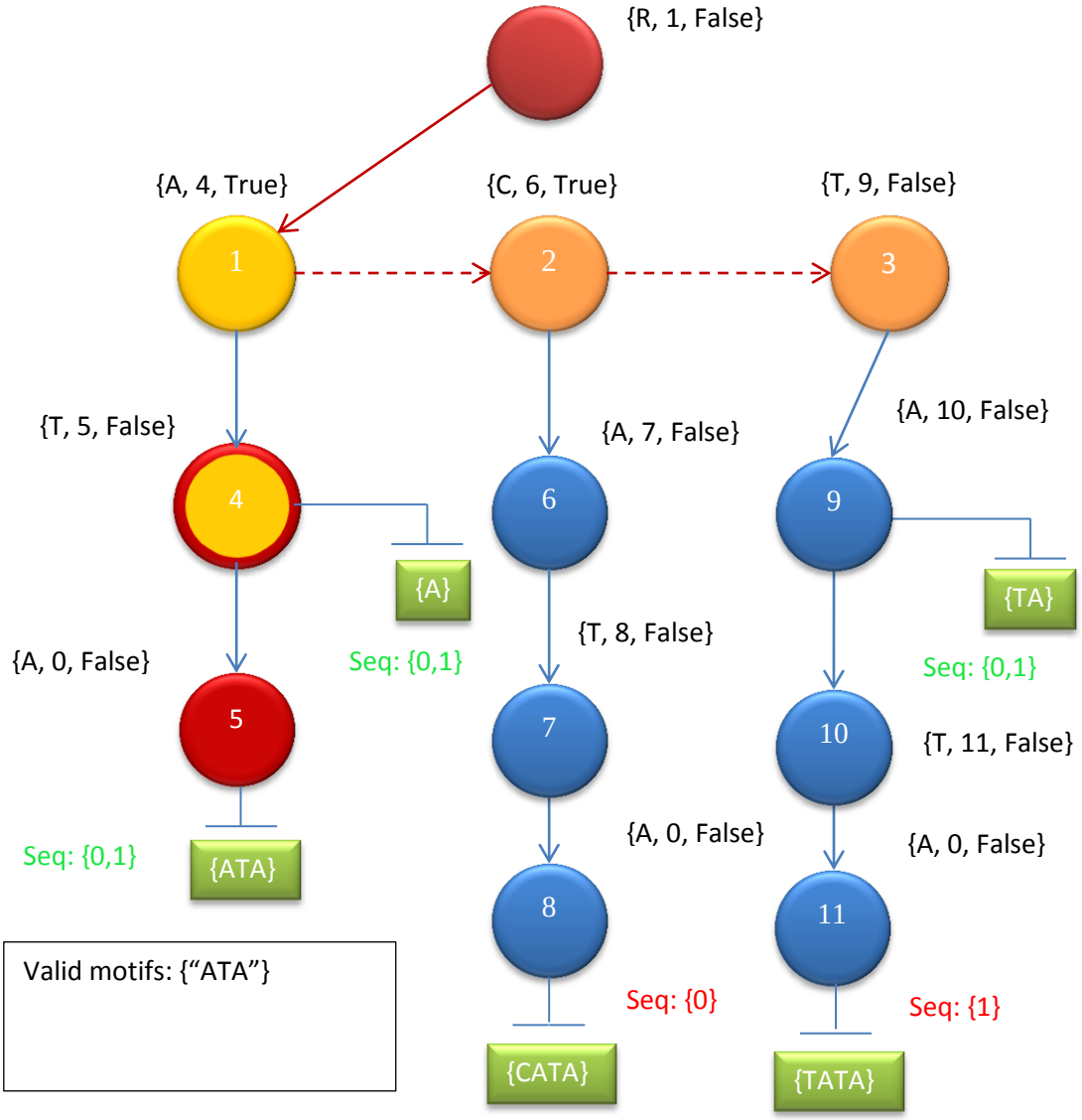

Figure 5-2 – Exact moment when node 5 is being processed. In red, nodes already processed. In yellow, nodes visited. In yellow with red ring, current node being processed. In green, sets with a valid number of sequences. In sets, nodes with an invalid number of sequences.

After adding the motif “AT” to the list, the control of the execution returns to node 1, and motif “A” is discarded after being checked because minimum length requirement.

It is time to move forward to the next sibling 2, from where the execution of the algorithm continues until all the nodes are visited.

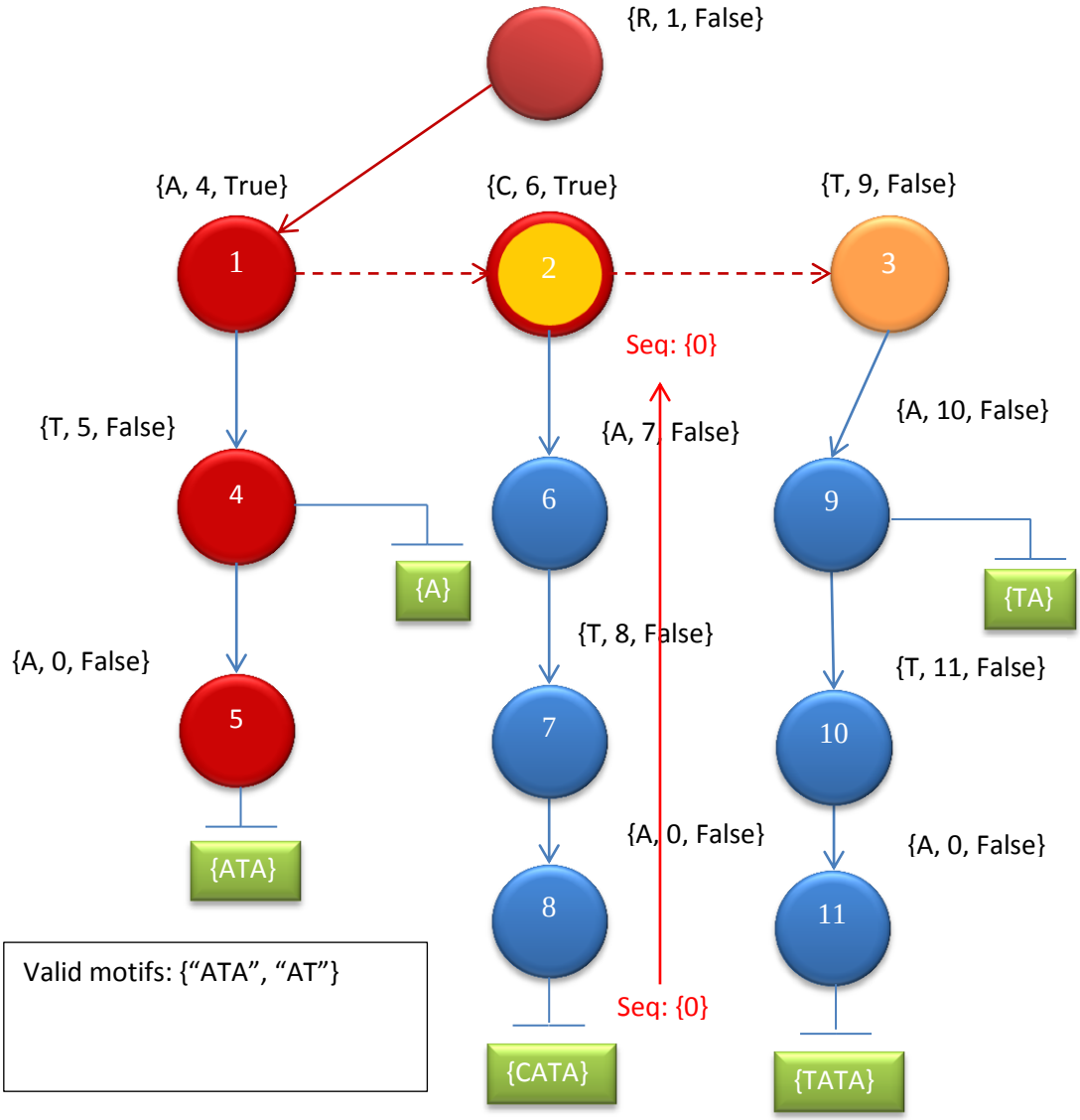

Figure 5-3 - Exact moment after all the motifs starting with "A" have been visited. Current node being processed is node 2. The sequence information is returned by node 8 to node 2.

During the execution we need to know how many sequences occur in the current path. This is immediate if it is a splitting or leaf node, but not for non-splitting nodes because they are not stored in the additional structure. When we are visiting a non-splitting node we will march down the path even if we exceed the maximum length needed, until we reach a splitting node or a leaf. Using our example, we will visit the node 8 even if the length of the motif is 4 and it is bigger than the maximum allowed. The

reason is that we need its *Sequence Information* to be sent to its father node since node 7 it is neither a splitting node nor a leaf but its length is correct.

In this case, in node 7 we find the motif “CAT” which has valid length, but node 8 returns as a *Sequence Information* that in this path only one sequence occurs, what means that the motif “CAT” is not valid.

When we reach a split node or a leaf, like node 8 that does not satisfy the quorum constraint, we stop the iteration and we return the sequence information of this node to its father.

At the father we will have two possible cases after a call:

- Father is a non-split node -> we will only receive the sequence information from its only child
- Father is a split node -> we do not need to use the information returned by the children because we can directly lookup for this information in the additional structure

When we are visiting a node where the pattern only appears in one sequence, we stop the lookup in this branch due to the branch and bound condition. But if the node is still promising (more than one sequence), we keep iterating through this branch until we reach a leaf or the branch and bound condition is reached.

When a motif with a valid length is reached we store it as a candidate motif. It means that if no degeneracy is allowed, only the exact motifs which appear in the trie are accepted, as for example: “ATA”, “AT”, “TA”.

After the execution we will get as a result a set containing all the different motifs with length between *k-min* and *k-max* with degeneracy *S* equal to 1 that appears in at least *q* sequences.

5.1.1. Validation of the exact motif discovery algorithm

We will use the sliding window approach in order to check the correctness of our motif discovery algorithm. After the execution of our algorithm, we return a set containing all the motifs of length between *k_min* and *k_max* that appears in at least *q* sequences.

To check if those results are correct we will use an exhaustive algorithm using sliding window that consists in scanning all the sequences varying the window size between *k_min* and *k_max*.

We create *q* sets where storing the substrings without repetitions. It means that we will try to insert the substrings with length between *k_min* and *k_max* in the first set. If the substring does not appear in the set, is added to that set. When the substring already appears in the set, we try to add it in the second set where we check again if the substring already appears in it, and we repeat this until the set number q. If we add a substring in the *set_q*, it means that this substring appears in *q* sequences so it is a valid motif.

At the end, we only need to compare the *set_q* with the set returned as a result of the motif discovery algorithm setting the quorum equal to *q*.

With this process we can ensure the algorithm is correct.

5.2. Degenerated Motif Discovery Algorithm

An important feature of a motif discovery algorithm is to detect degenerate motifs. It means we can find motifs that appear in several sequences while allowing some mutations.

The algorithm will search for motifs using the same parameters as exact motif algorithm needs: length between *k_min* and *k_max*, a minimum number *q* of sequences in where the motif occurs and now, a maximum degenerate value *S* that will allow a bigger number of motifs as we explained in chapter 3.

If before the valid motifs in our example were: “ATA”, “AT”, “TA”, now if degeneracy equals to 4 is accepted, it means that in addition to the first ones, also these motifs will be accepted: “ATN”, “ANA”, “AN”, “NTA”, “NT”, “NATA”, “NAT”, “NA”, where N is any character of the alphabet we are using. We will explain it in detail in the next chapter.

The idea of the algorithm is pretty similar as the exact motif discovery algorithm, but in this case we use a top-down approach adding first the shortest motifs and the algorithm will expand nodes situated in different paths.

For that reason, the alphabet used in this execution also contains a degenerate character. When trying to move forward from a node using the degenerate character, this implies that the execution will move through any of the existing paths from that node.

To solve this situation we will store in a vector all the possible sites that the algorithm will try to expand in further steps even if they are located in different paths. We can observe in the next figure how the algorithm, when searching for a motif starting with ‘N’ tries to expand the first visited nodes 1, 2 and 3. All the three nodes are possible motifs because from the parent using a degenerate character it is possible to move to those nodes, which are stored in a vector.

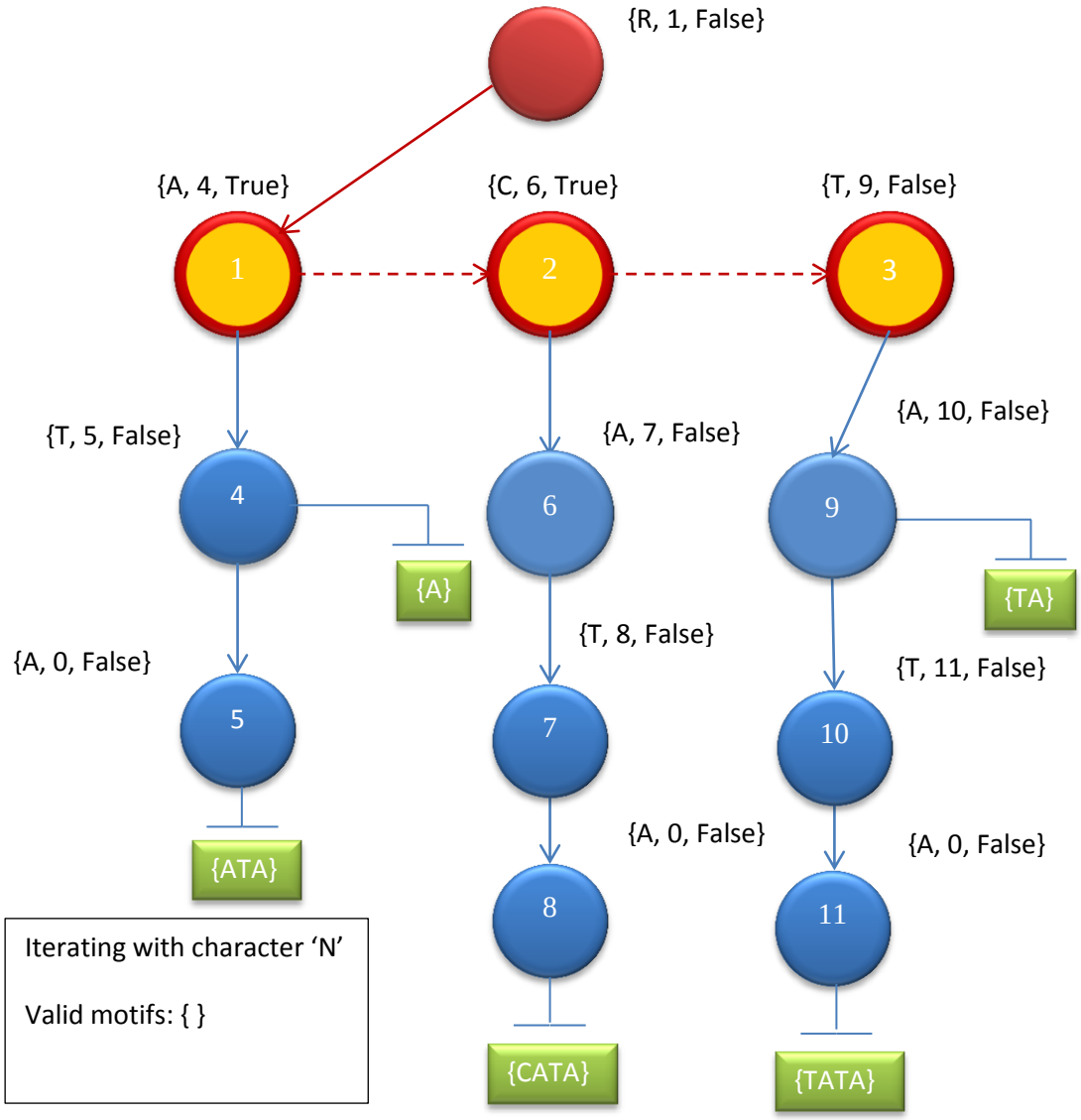

Figure 5-4 - First iteration using character 'N'. From root node it is possible to reach all its children.

Then we expand all the nodes stored in the vector checking first if it is a valid motif, and trying to move forward from all the nodes and adding one character to the current motif.

For the node 1, which motif is not accepted because its length, the algorithm cannot find a path using the character 'A', so the execution is stopped. But from nodes 2 and 3 it is possible to move forward using that character. For that reason, nodes 6 and 9 are stored to a vector and they will be processed in the next iteration.

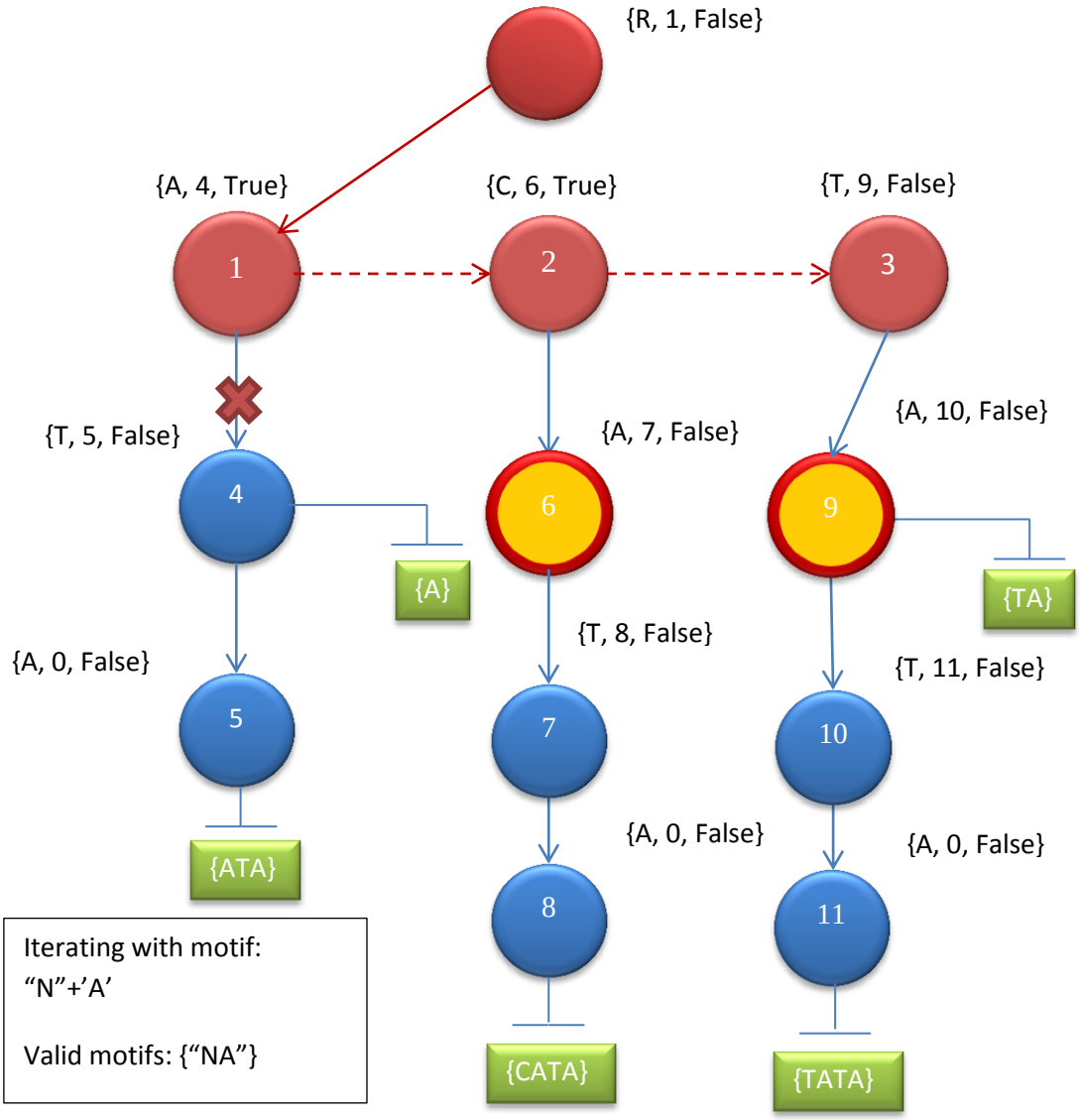

Figure 5-5 - Second iteration adding the character 'A' to motif "N". In red, motifs already processed for the current iteration.

At this point, the algorithm will check if motif "NA" is a valid one. In this case it is valid because it occurs in two sequences and its length is correct, so the motif is added.

To check if the motif is valid, we call a function for each of the nodes stored that returns us the number of sequences where this motif occurs. This process is very similar as in exact motif algorithm returning the Sequence Info if it is a splitting or leaf node or moving down in the trie until we reach one of those nodes. But now, since we can have different paths where the current motif occurs, we will join all the different sequence ids returned to know in which sequences the current motif with degenerate characters occurs.

After adding the motif "NA", the algorithm will try to move forward adding a new character: "A": "NAA". But from nodes 6 and 9 it is not possible, so the algorithm

removes the last character and repeats with the next character in the alphabet, 'T': "NAT", which is possible for both nodes 6 and 9. So in this case, motif "NAT" occurs in two sequences so it is added as a valid motif Figure 5-6.

We can observe easily the impact allowing degeneracy and the difference between this and the exact motif executions. In exact motif discovery is not possible to find this motifs since for each iteration with a single character only one path at a time can be expanded, and "CAT" or "TAT" only occur in one sequence each.

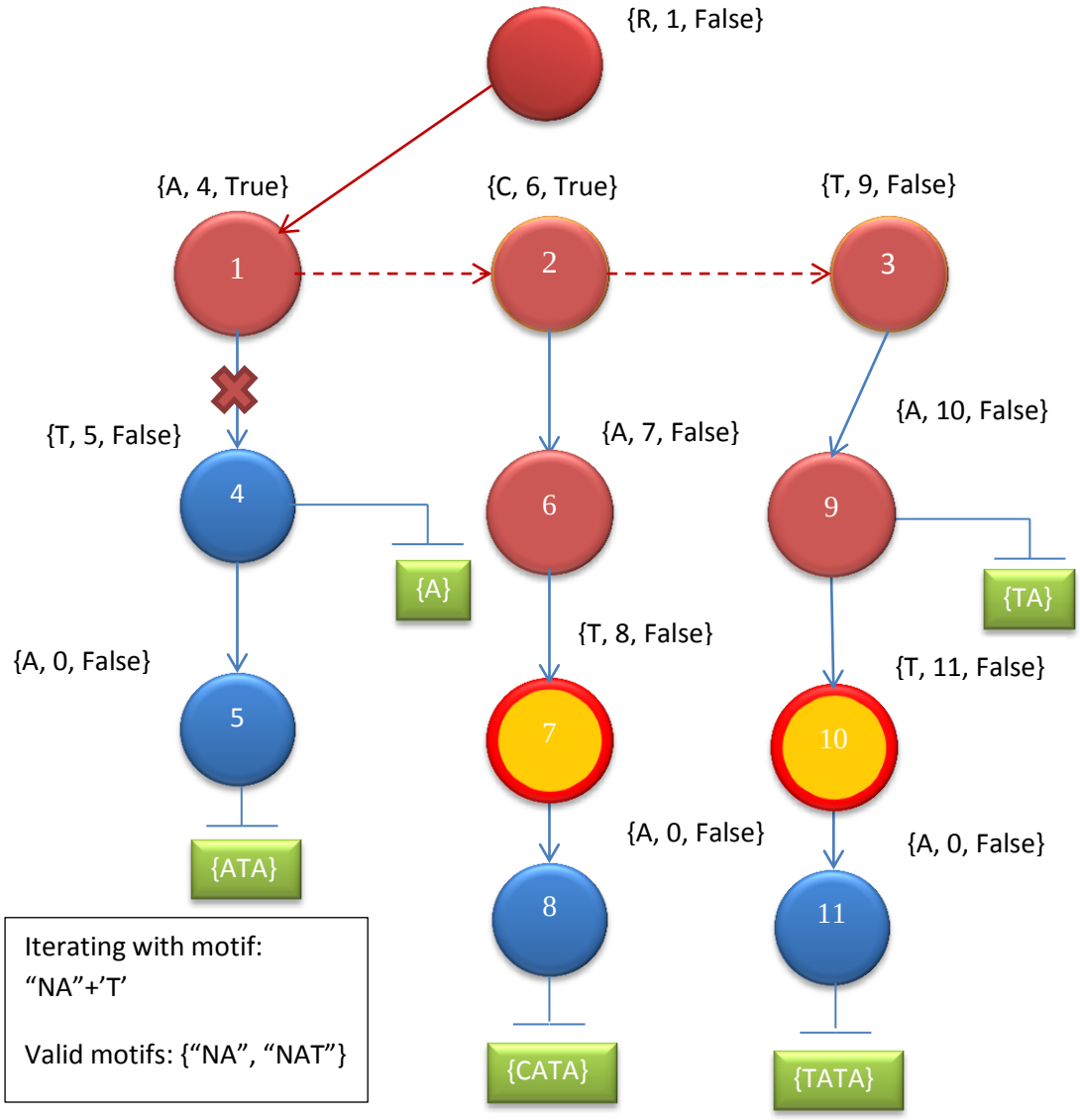

Figure 5-6 - Adding character 'T' to motif "NA". In red, motifs already processed for the current iteration.

For bigger values of degeneracy the strategy will be the same: expand from all the current reachable nodes through all the possible paths. And as the degeneracy allowed is limited, the algorithm checks every time that current degeneracy does not exceed the maximum.

The different sequences in which a certain motif occurs is harder to infer. We solve this problem as it follows:

In any point of the execution, the algorithm is processing a certain number of nodes *nNodes* between 1 and *S*. If only 1, it means that no degeneracy has been applied yet. But if *nNodes* is bigger than 1, the execution is following different paths. In this second case, to decide if the current motif is valid depending on the number of sequences, we will launch a query for each node and the list of sequences IDs that will be received as a response will be joined in the same structure. After all the reachable children have returned the list of sequence IDs that appear in their paths and joined in a set, the size of this set is the quorum which will determine if the motif is valid or not.

5.2.1. Validation of the degenerate motif discovery algorithm

We will use again the sliding window unit test to compare the results with the ones returned by the degenerate motif discovery algorithm. In this case we create *q* sequences and set the quorum between 1 and *q*, *k_min*=2 and *k_max*=16, and *S*=16. The result of this execution will return a set containing all the motifs with the specified properties.

For the sliding window approach, we create a function that receives the same set of sequences and first, searches for all the possible motifs without degeneracy. In the second step, the algorithm replaces one position of the motifs, starting from the first position and ending in the last one, the character for 'N'. This will return all the valid motifs with S=4. In the last step, we will repeat the same, but adding two 'N' along the sequence for those motifs with S=16. So we will have all the motifs with degeneracy 16 for the current sequences.

For the three different steps, we use the same sets which will help us to avoid repetitions, and after the execution we only have to compare if both size sets returned by the sliding window approach and degenerate motif discovery algorithm are equals.

6. Benchmarking

After the validations we would like to benchmark the structure. We will look at the construction algorithm performance, memory distribution and we will analyze the motif discovery algorithm.

We will execute different simulations setting different values for the parameters which are involved in the Suffix Trie performance, depending on which property we want to measure.

Using the results of those simulations, we will try also to deduce a general performance for the trie.

We ran the simulations on an Intel Core i5-2410M @ 2.30GHz laptop, with 4GB of RAM.

With all the information we can extract from those simulations we will also try to deduce which strategy is better for a given use case. We will analyze the 6 use cases of orthologous gene families. Those gene families contain genes with equal function in different organisms. We will test which choice is better: having 1 big trie or having F small tries, where F is the number of gene families and estimate memory and time requirements.

We will consider 6 cases, each time with families with 10 sequences of length 1000, and with the following numbers of families: F = 10, 100, 200, 1000, 3000 and 20000.

6.1. Construction Benchmark

One important aspect in the performance of the algorithm is the time spent in building the trie which depends on the number of sequences, their length and the *cutoff depth* of the trie.

In the next pages some plots are presented showing information related with the Suffix Trie construction algorithm.

The first plot (Figure 6-1) shows the Suffix Trie's total construction time versus different sequence lengths and depths. We can observe that all the lines have the same behavior depending on the amount of data or the depth we want to reach. It is important to note that the behavior is totally linear in number of sequences and their length, and complexity is *O(kNn)* as expected.

For the simulations with smaller values of depth cutoff we can observe that the time increases faster than for bigger values, while for the lines for depths between 13 and 16 they appear very close.

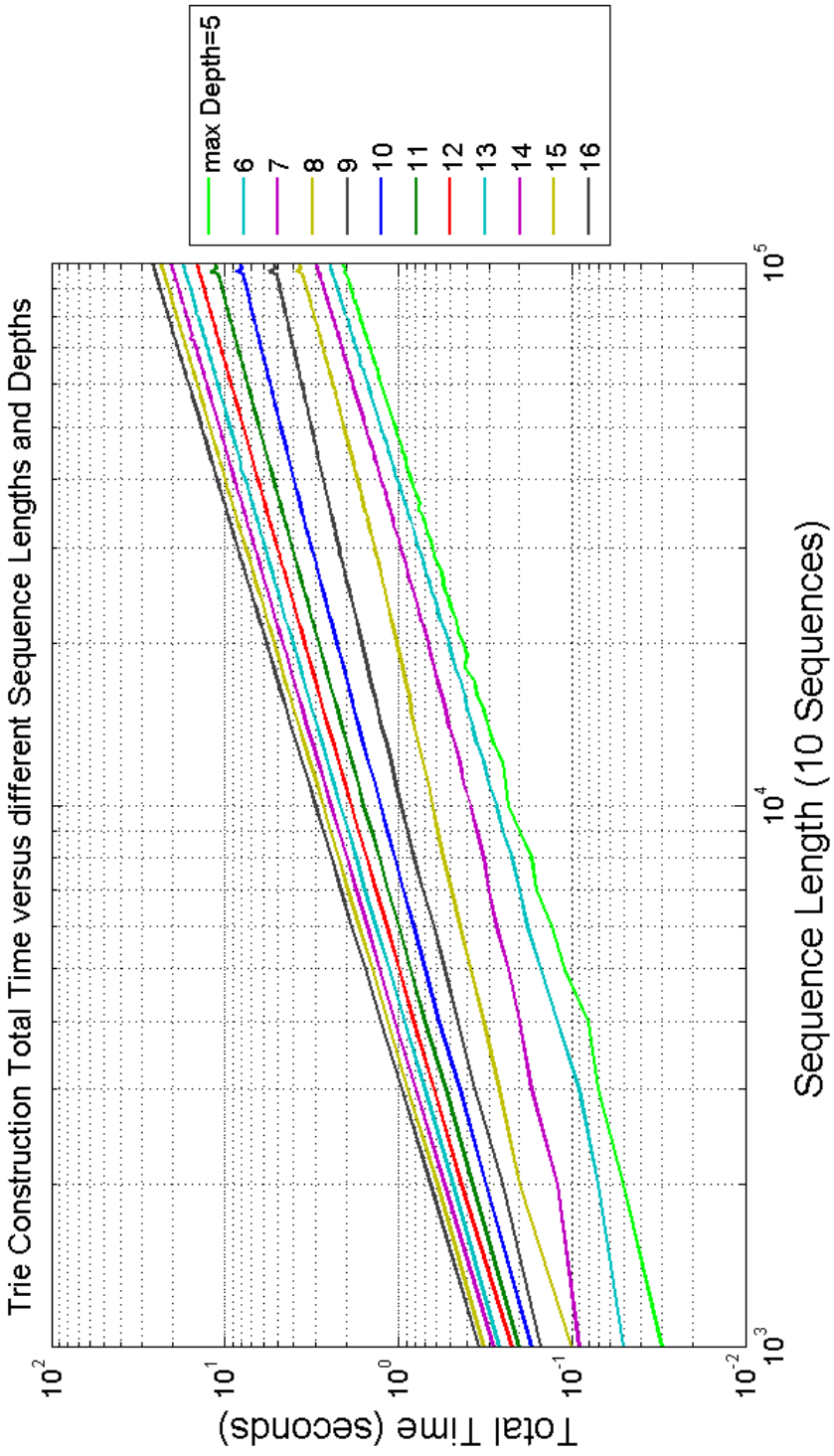

Figure 6-1 - Trie construction Total time for different depths and 10 sequences with lengths between 5 and 3000

The depth only influences the factor of the complexity, but it is bigger when we are building the trie until a small value of depth.

If we compare depth 15 with depth 5 we could expect a factor of 3 for the construction time, but regarding to the plot generated by the simulation, we obtained a bigger factor, which actually turns out to be k^2.

Next plot (Figure 6-2) shows the dependence of the construction time on the *cutoff depth*. We see two different regions. The first one occurs when the *cutoff depth* is smaller than 10, and it is possible to observe how the complexity is around $O(k^2)$, while for *cutoff depths* bigger than 10, complexity is $O(k)$.

This implies that the work per level is not constant, and one possible explanation is that for *cutoff depths* smaller than 10, we have a very dense trie where most of nodes have two or more children, while for larger *cutoff depths* we obtain a sparse trie in where on average there is only one child per node. The total construction time is linear in k in this case.

This has probably to do with the overhead produced by the creation and iteration through the queues in which we sort the different suffixes depending on the current character.

When we have to expand on node through 4 different paths, it means that we have to create 4 different queues, and we will process each of them separately. It causes an overhead that could result in the quadratic behavior for small *cutoff depths*.

The second region appears linear because we have are creating and processing less queues and it produces a smaller overhead, so the performance is better and the complexity is linear.

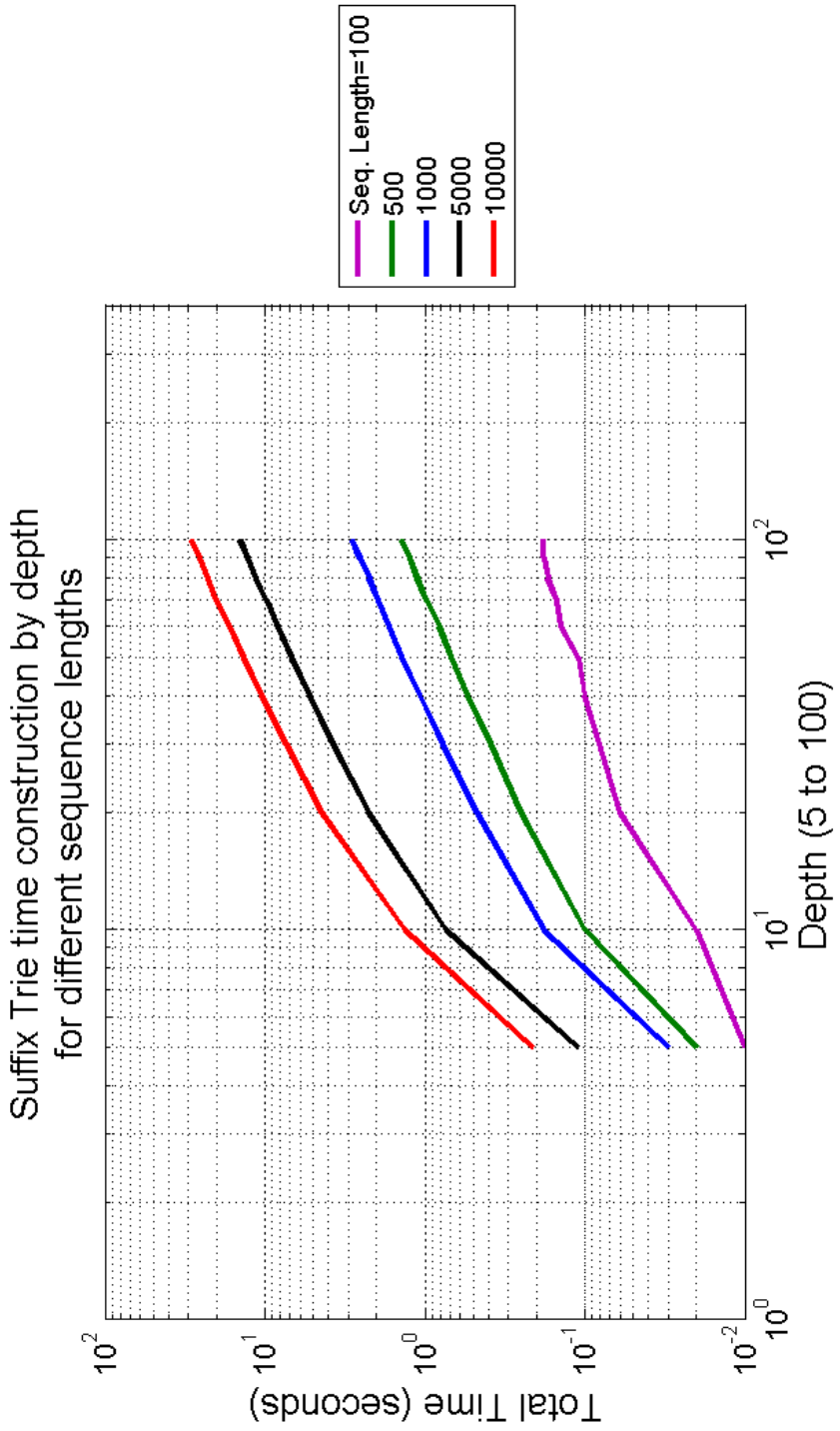

Figure 6-2 – Total Suffix Trie construction time versus depth for 10 sequences with different lengths.

6.2. Memory Distribution Benchmark

Now we want to analyze the suffix trie memory performance in terms of different properties like the number of nodes, leafs, distribution of nodes, etc.

Figures 6-3 and 6-4 contain the information relating to the number of nodes and the number of leafs for different depths and different sequence lengths.

In Figure 6-3, we can observe that for depths 5 and 6, the trie is full and the lines drawn in the plot are completely flat for all the sequence lengths. The reason for this is we have the maximum number of children for each node.

For depths between 7 and 10 it is possible to detect how those levels are growing in number of nodes until a certain point where the lines tend to flat. It means again that the maximum number of nodes that the structure can handle for this level is reached and it is not possible to create new nodes.

For depths bigger than 10, the lines grow during all the execution, and are completely straight, which means that the structure can handle still more nodes for those levels while its size grows linear.

Something similar happens for the number of leafs in the figure 6-4. It is expected that the number of leafs of the tries to be similar if we cut the trie at a certain depth. In this case, it is possible to observe how between 5 and 9 as a value of *cutoff depth* there is a variation of behavior after the simulations. For depth 5 and 6 the line is completely flat because the number of leafs is already the maximum number of leafs allowed for that *cutoff depth*. And for *cutoff depths* between 7 and 9, while increasing the sequence length the number of leafs increases until some point where starts to flat, adopting the same behavior than for the smaller depths we saw before.

For bigger *cutoff depths* we can observe how the number of leafs is pretty similar between them. The reason is because the trie is not very dense, so almost all the nodes are a chain of nodes with only one child. It results in that we will have almost the same number of leafs for depth 14 than for depth 15.

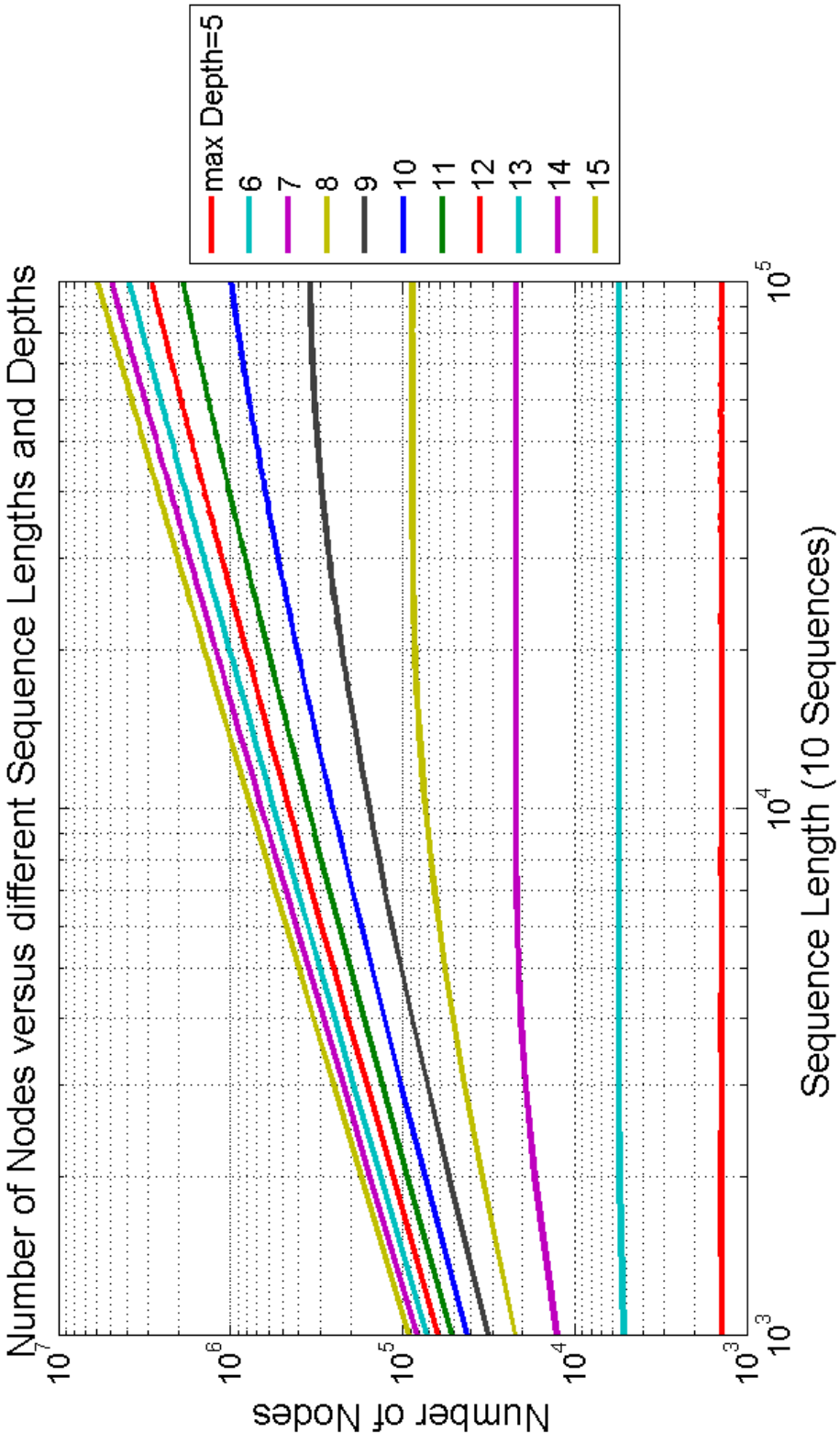

Figure 6-3 - Number of Nodes for 10 sequences with different lengths and depths

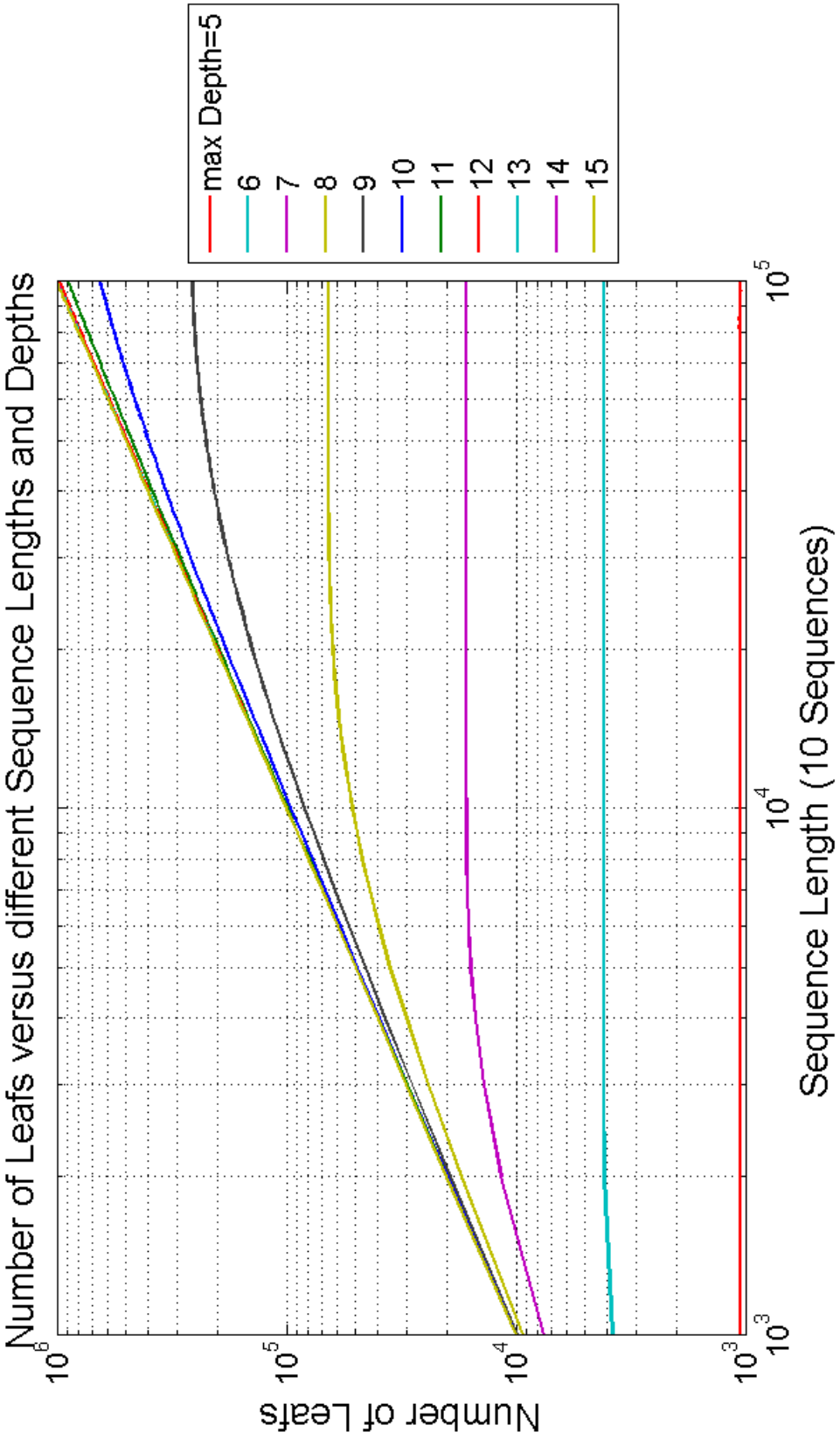

Figure 6-4 - Number of leafs of 10 sequences for different lengths and depths

Figure 6-5 and 6-6 shows an important feature of the Suffix Trie which differs from the Suffix Tree. In the Suffix Tree there are no nodes with only one child, being the reason that the Suffix Tree collapses the branch when it has only one child.

But in the Suffix Trie, and especially when we choose a big value for the *cutoff depth* (in this case=15), it is quite common to have branches that are a chain of nodes with only one child. This happens when from a certain node there is only one unique suffix to be reached. Because of this, the nodes with only one child are the most numerous in the trie for large *cutoff depths.*

All of them grow linear as we showed in Figure 6-3, and while the number of nodes with 3 or 4 children is similar during all the simulation, the number of leafs is smaller than the number of nodes with only one child. The reason is because for depth=15 it is not possible to have so many long different suffixes and a lot of branches are occupied only by nodes with one child.

If the Figure 6-6 we set the *cutoff depth* for the simulation to a smaller value = 9 and we repeated the simulation with long sequence lengths, to observe how the number of nodes with only one child tends to be 0 as well as the nodes with 2 and 3 children, while the nodes with 4 are the dominant ones. This is because the trie becomes full.

The number of leafs will tend to flatter as soon as we reach the maximum number of leafs allowed. The maximum number of leafs is 4^k.

In our plot, the number of nodes with 5 children appears with almost a straight line which is growing very slowly. For 10 sequences of length 100, there are only 23 nodes with all their possible children. And at the end of the simulation, with length=10000, there are 55 nodes with 5 children. And this number will hardly grow during all the execution if we increase even more the length of the sequences. It occurs because to become a node with 5 children, it has to contain as a child all the possible character plus the sentinel character of the sequence. And for sequences of length n, we will have only n suffixes. So it depends on the number of sequences and not in the length of the sequences.

In a simulation with 10 sequences, in the first level we will have 4 nodes with 5 children, but in the next level, the number of nodes with 5 children is limited to 10, because it is not possible to have more than 10 suffixes of length 2 if there are only 10 sequences.

If we set the *cutoff depth* to 15, and considering that the first level can only have 4 children, it means that we will have 14 levels with 10 nodes with 5 children, and the first level only 4, what results in a maximum number of 145 nodes with 5 children.

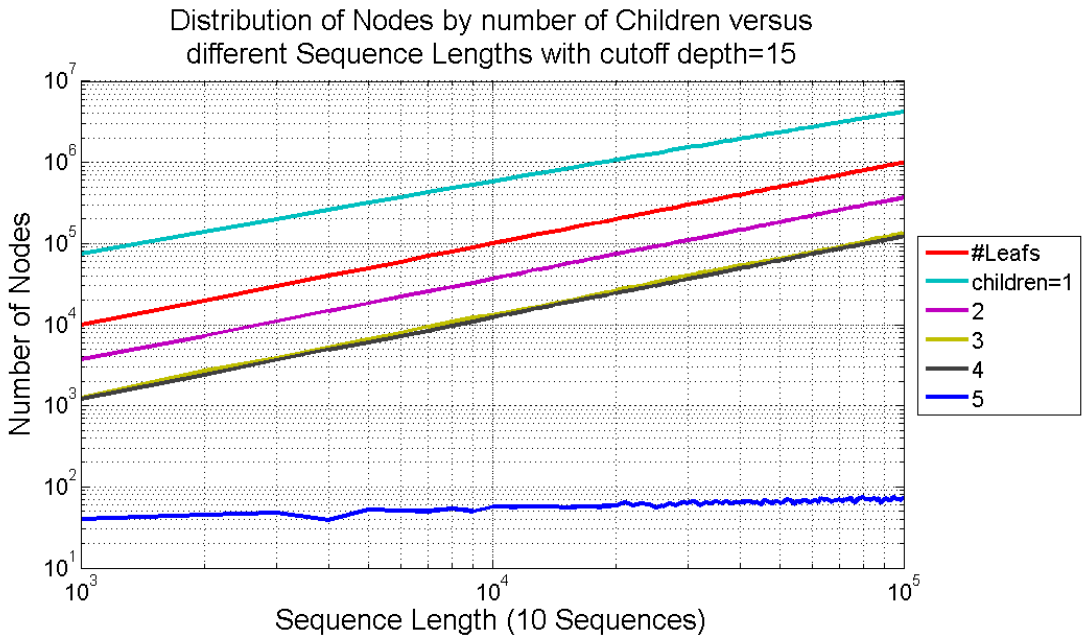

Figure 6-5 - Distribution of Nodes by Number of Children for 10 sequences with different lengths and different depth values

Looking at Figure 6-6, for sequence lengths = 100 the number of nodes with 1 child is almost 3 times higher than the number of leafs, while starting from length = 3000, their number quickly drops meaning the trie and tree become very similar.

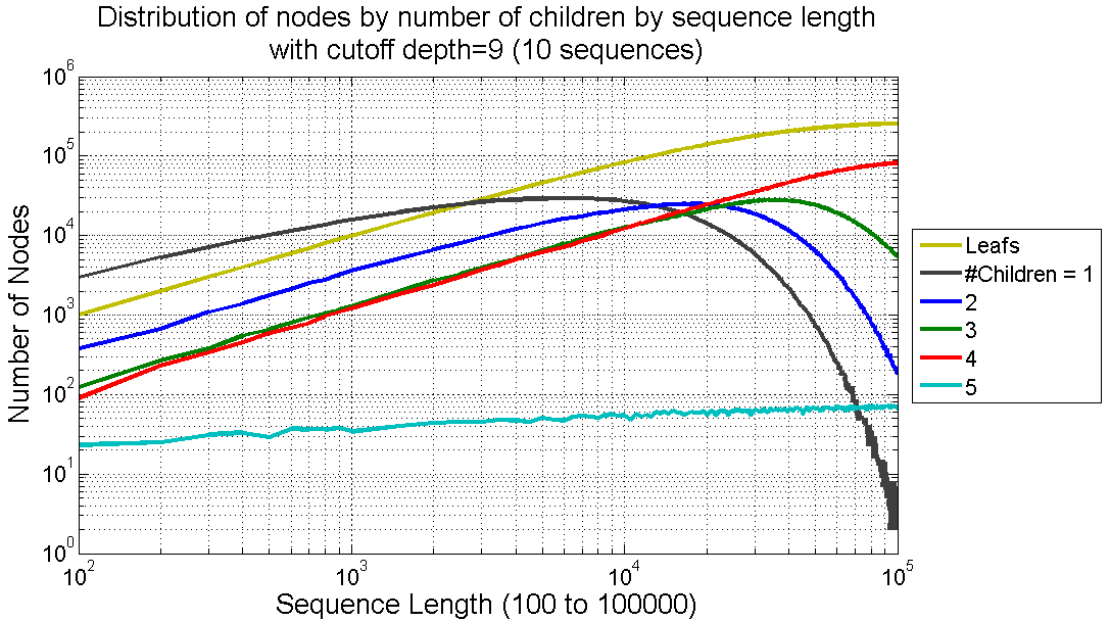

Figure 6-6 - Distribution of nodes by number of children versus different sequence length and *cutoff depth* = 9

As we have shown for the number of leafs, the plot with the number of splitting nodes is very similar, and is repeating the same behavior for depths over 10 and all of them have almost the same number of splitting nodes and slopes. It means that the number of splitting nodes grows very similar for all of the levels in the trie until one level begins to become full. Then, the number of splitting nodes grows slowly until the maximum possible number of splitting nodes is reached, which is 4^k where k is the level on the trie (Figure 6-7).

The number of splitting nodes for small depths is the same again during all the execution and the reason is the same one: the trie is full and it is not possible to have more splitting nodes, since all the nodes in the trie are splitting nodes.

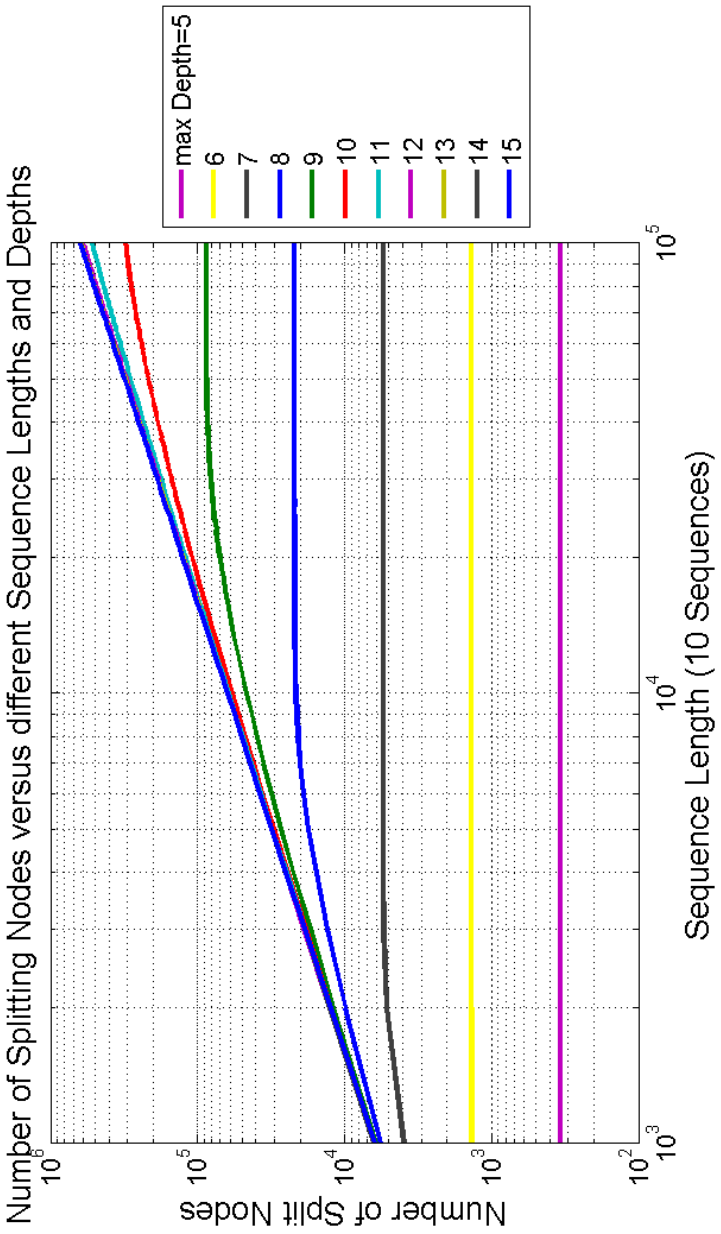

Figure 6-7 - Number of splitting nodes for 10 sequences with different lengths and depths

We now present a plot (Figure 6-8) that shows the number of splitting nodes and leafs per level in the Suffix Trie. The first observation we can make is that the number of nodes for the first levels is constant for the complete simulation. This can be explained because the trie is completely full at those levels for the configuration used (number of sequences from 5 to 3000, of length 1000).

Something interesting is to figure out how the levels become completely full, and their curve stops growing, crossing the lines of deeper levels that keep growing up. We can see this behavior for lines of depths 6, 7, 8 and 9, and also we can expect that the line for depth 10 will stop growing and will cross the line in the plot for depth 11.

For levels completely full, the value when the line tends to be flatter can be calculated as 4^k. With k=7, for example, the maximum number of nodes is 4^7 which result is 16.384 nodes. With a small number of sequences, there is a bigger number of non-splitting nodes, but the line behaves linear until it reaches the 16.384 nodes. It means that the level does not contain non-splitting nodes, all the nodes are leafs or splitting nodes.

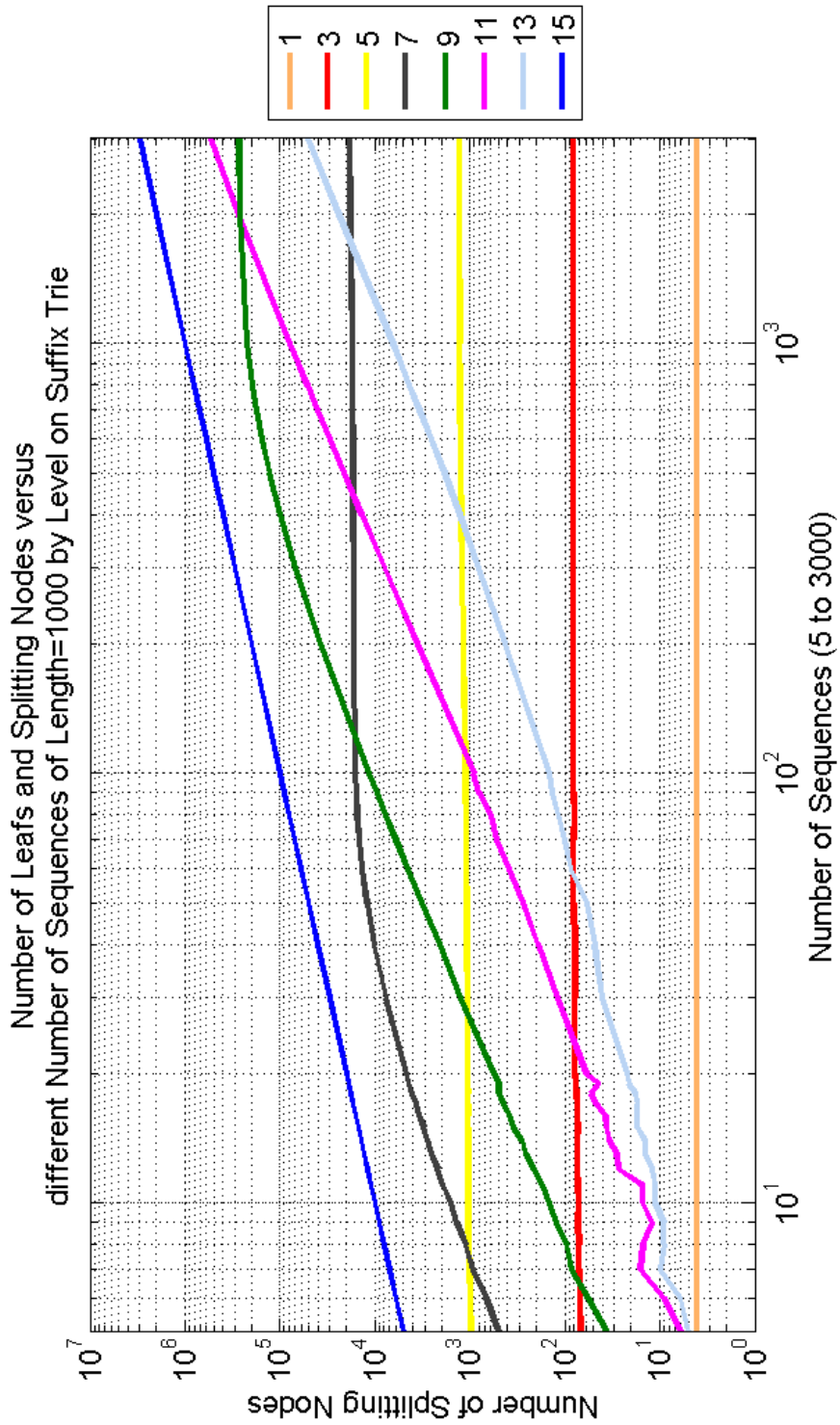

Figure 6-8 - Number of Splitting nodes by number of sequences of length=1000

6.2.1. Suffix Trie and Suffix Tree comparison

Until now, the plots presented the number of leafs and splitting nodes, but another important fact in the Suffix Trie are the non-splitting nodes, that as we explained are the nodes with only one child, which do not appear at the Suffix Tree because it collapses those nodes into the same edge. We will compare both structures in terms of memory and time consumption.

First we show in Figure 6-7 the plot that shows how the number of non-splitting nodes increases. In the simulation we were using 10 sequences of length 1000, and we were increasing the depth from 5 to 700.

We can observe a fast increasing of the number of this kind of nodes for small depths, a lot of splitting nodes which makes grow the number of non-splitting nodes. But early the line changes its behavior and the slope is less steep and linear, meaning that no splitting nodes occur from that level on.

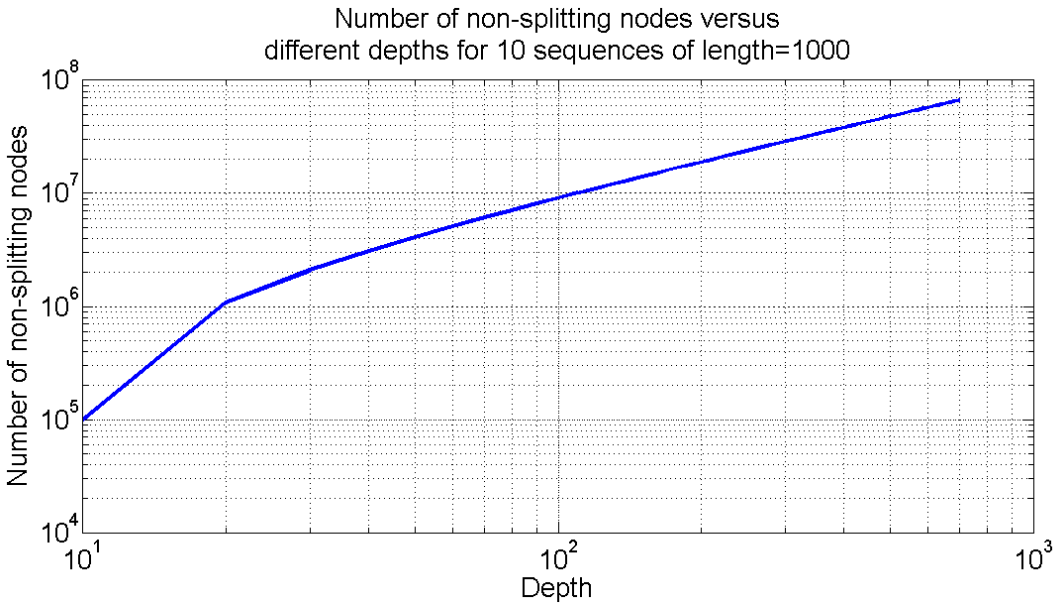

Figure 6-9 - Number of non-splitting nodes for 10 sequences of length 1000 for different *cutoff depths*

As we explained in chapter 3, Suffix Tree requires 44 bytes per character versus the 16 bytes required by the Suffix Trie. We want to find the intersection between both structures in regard to memory efficiency.

In Figure 6-10, for 10 sequences of length 1000 and varying the depth, we can observe how the intersection is located around *cutoff depth* = 10.

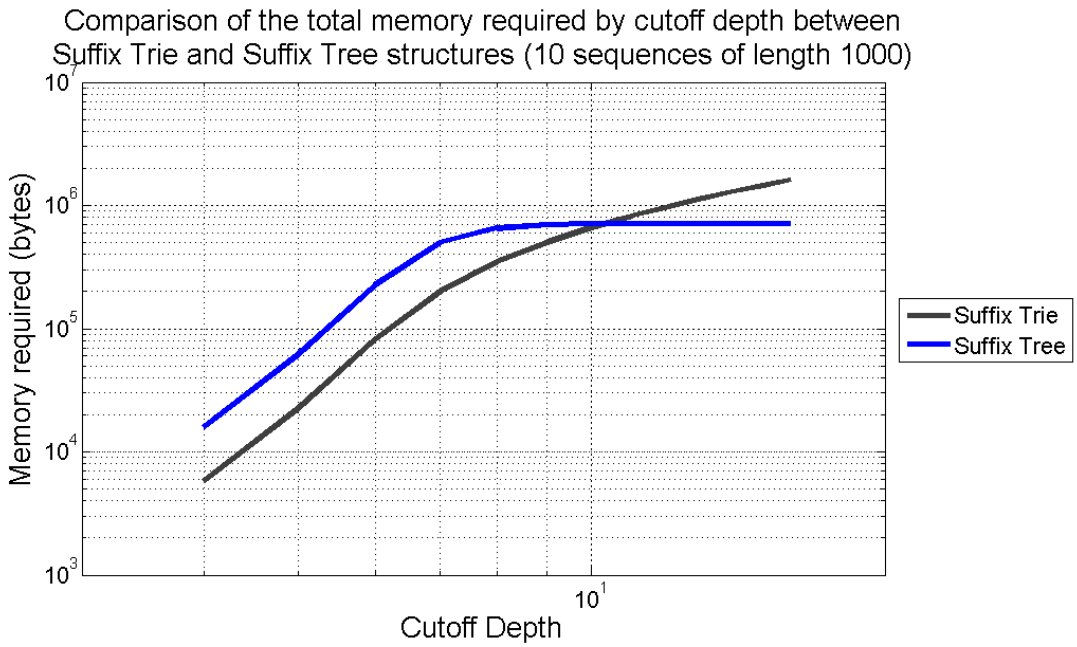

Figure 6-10 - Memory required by different values of cutoff depth for Suffix Trie and Suffix Tree, for 10 sequences of length 1000.

For small *cutoff depths* both structures are dense, which keeps the number of non-splitting nodes low and the difference of memory required by character makes the Suffix Trie more suitable.

When the *cutoff depth* increases we obtain a sparse trie, what causes a rising in the number of non-splitting nodes at the Suffix Trie, which will have the slope of figure 6-9.

At the Suffix Tree these new non-splitting nodes are collapsed into a unique node. This explains why for bigger *cutoff depths*, the memory used by the Suffix Tree remains stable: all the paths to expand belong to a unique suffix and new splitting nodes will not appear.

We can conclude that the Suffix Trie is a good choice regarding to memory requirements when the trie is dense or the cutoff depth is small. But it requires a big amount of memory for sparse tries, and the Suffix Tree's property which collapses the nodes with one child provides a more efficient memory usage.

6.2.2. Orthologous gene families use cases

The average set sizes in the Suffix Trie will help us to discuss which strategy is more suitable: handling one trie containing all the sequences or building a small trie per each orthologous gene family.

First we are going to analyze the set sizes of the leafs and splitting nodes of the Suffix Trie sorted by level (see Figure 6-11). In order to obtain this plot, we configured the simulation starting with 5 sequences until 3000 sequences of length 1000, and using as a *cutoff depth*=15.

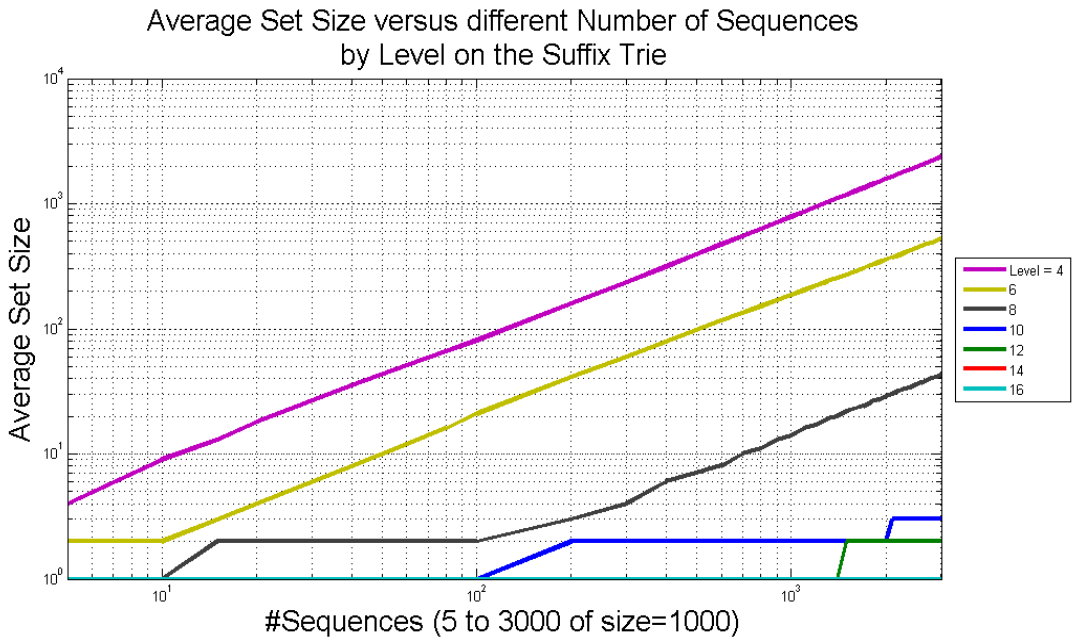

Figure 6-11 - Average Set Sizes for different number of Sequences and level on the trie

What we can observe are two different regions in the plot. In the first region, each level starts with only 1 sequence ID per level in average and while increasing the number of sequences, the average set size jumps to two where remains stable until a certain point from where starts to grow with linear complexity.

We can explain this behavior and the plateaus which appear in the Figure 6-11 using the a probabilistic approach based on the binomial distribution.

We will focus on level *k*=8. We will study the number of occurrences of all the *k_mer*.

First we have to calculate the expectancy, which is the number of times expected that a *k_mer* occurs in a certain sequence, and it is calculated with the following formula:

$$E(x) = n \cdot p$$

Being *n* the sequence length, and *p* the probability of the *k_mer* to appear in 1 sequence:

$$p = \frac{S}{|\Sigma|^k}$$

Where *S* is the degeneracy and $\boldsymbol{\Sigma}$ is the alphabet.

The expectancy for a *k_mer* of length 8 to appear in a certain sequence of length 1000 with degeneracy *S*=1, is:

$$E(x) = 1000 \cdot \left(\frac{1}{4^8}\right) = 0{,}015$$

The value 0,015 indicates the probability to appear in one sequence, then to estimate how many times will appear in a set of 10, 100 ad 1000 sequences:

$$\#k\_mer = 10 \cdot 0{,}015 \approx 0{,}15$$

$$\#k\_mer = 100 \cdot 0{,}015 \approx 1{,}5$$

$$\#k\_mer = 1000 \cdot 0{,}015 \approx 15$$

We can check in the Figure 6-11 how the binomial probability gives us a good estimation of the quorum expected for a *k_mer* in a set of sequences.

These set size averages will help us to deduce which strategy is better in terms of memory efficiency: creating a big trie or several small tries.

In our use case we will set 10 sequences per family of length 1000, and the following numbers of families: *F* = 10, 100, 200, 300, 1000 and 20000. We obtained the results for the smaller gene families from simulations, but for a large number of families is not possible to run a complete simulation because of memory limitations, so we will extrapolate for these use cases.

For estimate the memory consumption of both strategies we will use the Figures 6-3, 6-4, 6-7, 6-8 and 6-11 and their source data.

We start with the approach that uses *F* small tries. Only one simulation for 10 sequences of length 1000 is required, and the results will be extrapolate for the rest of use cases.

Calculating the memory required is divided into two measurements: the average set sizes and the total number of nodes.

With the sum of the different average set sizes we will obtain the number of sequence IDs we need to store into the *Sequence Information* structure.

We will need to sum this value with the total memory occupied by the nodes of our trie. We start showing the measurement for creating one small trie containing one unique family.

For *F*=1:

Total number of nodes: 100.721
Total number of leafs: 9.990
Total number of splitting nodes: 6.232
Total number of non-splitting nodes: 84.499
Total size of the sets: 31.904
Construction time (sec): 0,43

With this information we can calculate the memory required since each node occupies 16 bytes and set contains integers:

$$Memory\ required\ =$$

$$\#nonsplitting \cdot 6\ bytes + \#splitting \cdot 6\ bytes\ + \#leafs \cdot 16\ bytes\ + set\ size \cdot 20\ bytes$$

We require 1,2MB to store one small trie for one unique family, and the memory required for the other use cases can be easily calculated and are shown in Table 1.

Number of families	Memory required	Time required (seconds)
F = 10	12,8MB	4,3
F = 100	128MB	43
F = 200	256MB	86
F = 300	384MB	129
F = 1000	1,25GB	430
F = 20000	25GB	8600

Table 1 – Memory and time required in building F small tries, one per each gene family.

Now we will show how to obtain the requirements for the approach which creates one big trie with all the gene families. This approach requires a simulation of the construction of a Suffix Trie varying the number of sequences each execution.

We will calculate first the memory required due the total number of nodes, which is shown in Table 2.

Number of families	*#nodes*	Memory by number of nodes	#leafs	Memory by number of leafs	Total average set sizes	Memory required by sets	Total Memory Required
F = 10	$842 \cdot 10^3$	4,82MB	99747	0,95MB	$480 \cdot 10^3$	9,17MB	14,9MB
F = 100	$6,77 \cdot 10^6$	38,74MB	995703	9,5MB	$6,46 \cdot 10^6$	123,2MB	171,5MB
F = 200	$12,5 \cdot 10^6$	71,87MB	1990148	18,98MB	$13,9 \cdot 10^6$	265,1MB	355,9MB
F = 300	$17,9 \cdot 10^6$	102,81MB	2984097	28,46MB	$21,7 \cdot 10^6$	414,3MB	545,6MB
F = 1000	$52,3 \cdot 10^6$	0,29GB	9934560	0,09GB	$81,8 \cdot 10^6$	1,52GB	≈1,9GB
F=20000	$818 \cdot 10^6$	4,57GB	198213932	1,85GB	$2,36 \cdot 10^9$	44,03GB	≈50,45GB

Table 2 - Memory requirements for one trie containing all the gene family sequences with estimations for F=1000 and F=20000.

Number of families	Total Time
F = 10	3,31
F = 100	28,84
F = 200	54,2
F = 300	80,82
F = 1000	255
F=20000	4179

Table 3- Total time required for building one trie containing all the gene family sequences

We can compare now both approaches, results of which are shown in Table 4.

	F small tries		1 big trie	
Number of families	Memory	Time (sec)	Memory	Time (sec)
F = 10	12,8MB	4,3	14,9MB	3,31
F = 100	128MB	43	171,5MB	28,84
F = 200	256MB	86	355,9MB	54,2
F = 300	384MB	129	545,6MB	80,82
F = 1000	1,25GB	430	≈1,9GB	255
F=20000	25GB	8600	≈50,45GB	4179

Table 4 - F small tries vs 1 big trie approaches comparison in terms of memory and time efficiency.

As we can observe in the table, the total memory consumption when building F different tries is half of the memory required when all the families are in the same trie. That is caused because, even the bigger number of nodes created by the first approach, their sets contain less sequence IDs and since each sequence ID requires 20 bytes it has bigger impact in the total memory efficiency.

The approach that builds only one trie saves memory because a lot of nodes are shared by different sequences and there are less non-splitting nodes, but the ones which are created have to store sets with a lot of sequences IDs. When building a unique trie for 20.000 families, these sets require almost the 90% of the total memory consumed.

Number of Families	Non-splitting nodes memory (%)	Splitting nodes memory (%)	Leafs memory (%)	Sequence IDs sets (%)
F = 1	37,8	2,7	12	47,5
F = 10	26	2,4	10,2	61,4
F = 100	17,2	2	8,9	71,9
F = 200	15	2	8,5	74,5
F = 300	13,8	1,9	8,4	75,9
F = 1000	10,6	1,8	7,8	79,8
F = 20000	5,5	1,4	5,8	87,3

Table 5 - Percentage of memory required by the trie sorted by non-splitting and splitting nodes, leafs and sequence IDs sets.

We can observe also that building a unique trie is twice as fast than building F different tries. That has sense since as shown in Figure 6-2, the overhead produced building the first levels of the trie is larger than for deeper depths.

Another conclusion we can observe from the comparison is in the average size of the sets in leafs and splitting nodes of both approaches. Using the results for F=1 and F=300, we will compare their average set sizes (Table 5). To compare both cases, we will multiply F=1 by 300.

	F = 1*300	F = 300
Leafs and Splitting Nodes	$4{,}86 \cdot 10^6$	$4{,}83 \cdot 10^6$
Non-splitting Nodes	$2{,}53 \cdot 10^7$	$1{,}31 \cdot 10^7$

Table 6 - Comparison of the number of splitting and non-splitting nodes in a trie with 300 families and the extrapolation of one small trie with 1 family. Each family containing 10 sequences of length 1000.

This table shows an interesting property of both approaches, which is that the number of nodes in the unique and big trie will be always smaller than in F small tries. As we explained, in a dense trie the number of splitting nodes tends to be bigger than the non-splitting nodes, and these nodes are shared by more than one suffix which saves memory.

In a small trie the number of non-splitting nodes is bigger since for large suffixes, few of them will share the same path and it requires the creation of a big number of nodes with only one child.

On the other hand, these non-splitting nodes do not need to store sets with the *Sequence Information*. We will analyze if it has a big impact.

Comparing the memory required to store the nodes (102,81MB) and the required for the sets (414,3MB) for F=300 in Table 2, shows that the impact of the size of the sets is bigger than the impact caused by the total number of nodes.

We can deduce for the case we are working with a huge number of sequences, that the approach that creates F small tries is much better since we will save a lot of memory not storing the sets, even if we have a large number of nodes.

When we work with few sequences but very large, the approach that creates only one big trie will be better, since the number of sequences IDs to store will be little, and the number of nodes will remain lower in comparison with the F tries.

6.3. Exact Motif Discovery Algorithm Benchmark

We will present now some graphs to show the performance of the exact motif discovery algorithm.

The first plot represents the total number of motifs per length. We can observe that we reach a peak for motifs of length 8, since the motifs with smaller length are less in number. And for larger motifs, they appear less time in two or more sequences.

The explanation for that is because for short sequences there are less motifs than for the bigger ones. When we are finding motifs, we have a worse frequency with short motifs even if we found all of them. In the other hand, finding longer motifs, the performance will tend to be badly as the motif size increases. The explanation for that is because there are less motifs repeated among all the sequences (quorum constraint), so the algorithm will need a lot of time to iterate through the trie, but because of the small number of motifs, and the number of queries because the branch and bound strategy, will turn out in a bad performance.

Regarding to the plot, we will obtain the best performances when looking for motifs between 7 and 9. The explanation for that is because the number of possible motifs is big enough, and the overhead produced by the iteration through the trie is still not very big.

The distribution of motifs follows again a binomial probability, and for this data the biggest number of appearances occurs for lengths between 7 and 9.

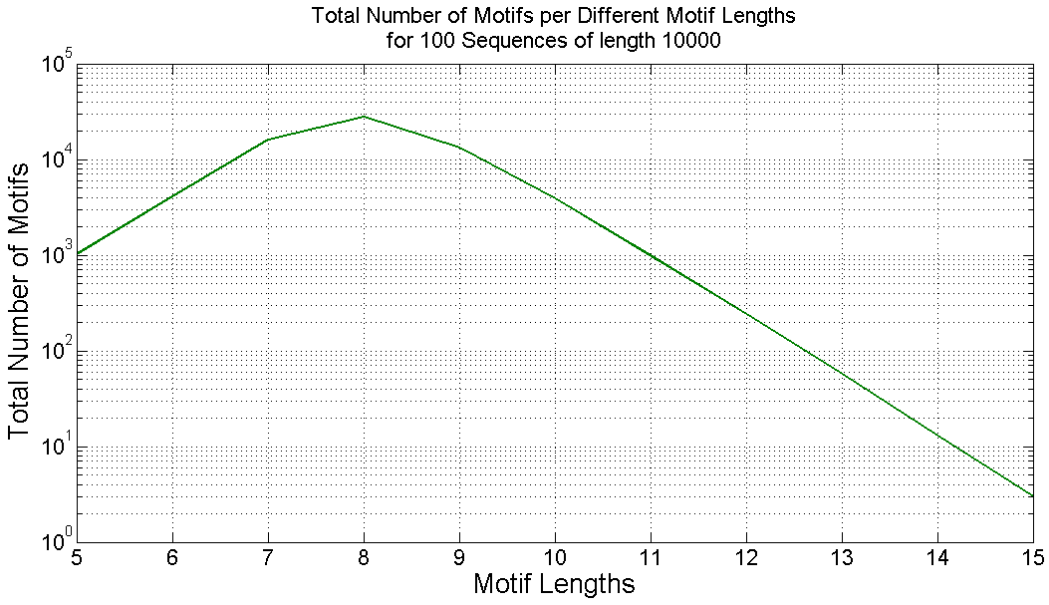

Figure 6-12 - Total Number of motifs by length, for 100 sequences of length 1000, and cutoff depth 16

Another property we want to observe is related with the number of motifs we can find if the value of the quorum changes. For that reason we did a simulation where for 100 sequences of length 100.000, the quorum was varying its value between 2 and 100.

It is possible to see that the quantity of motifs decreases while the value of the quorum increases as expected.

As we calculated before for the average set sizes in Figure 6-11, the plateaus are caused by the binomial probability of these motifs occur for the number of sequences of length 10.000. The results are shown in the Figure 6-13.

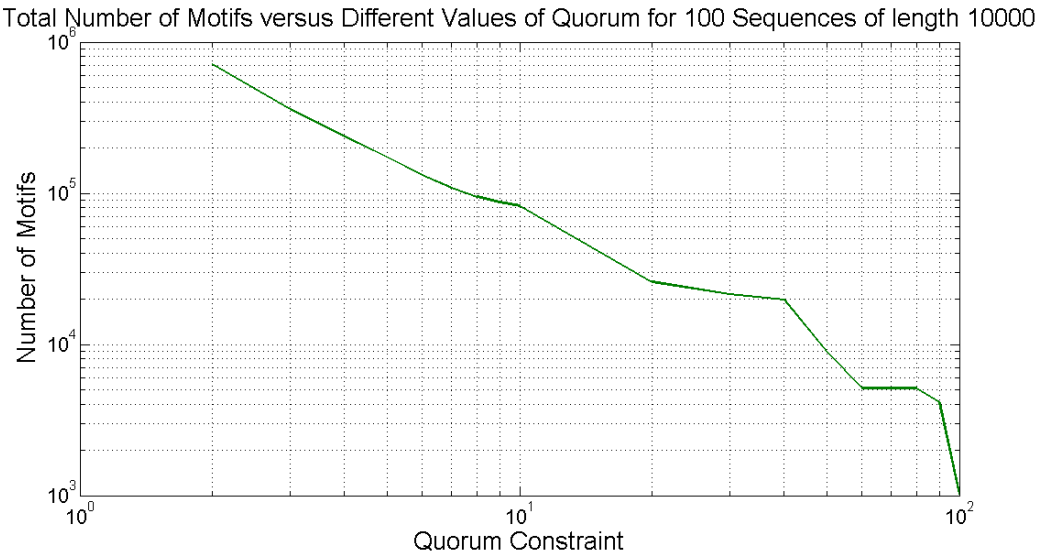

Figure 6-13 - Total number of motifs by different values of quorum, for 100 sequences of length 10000.

6.4. Degenerate Motif Discovery Algorithm Benchmark

The next figure presents the total time consumption by the motif discovery algorithm for different degeneracy allowed and varying the quorum constraint. It is expected that for bigger degeneracy the time consumed is bigger, and in this case we can conclude that it is also proportional to the number of degenerate characters introduced.

Also, the bigger the quorum value is, the faster the execution of the algorithm will be. For small values the algorithm is slower because the branch and bound uses the quorum value to stop the execution and in this case it will not stop the execution. It means that for a Suffix Trie if we set the quorum to 1, the algorithm will iterate until it reaches the leafs which results in a huge time consumption.

But when the quorum value is large, the branch and bound condition will stop the execution through the branches in the first levels of the trie. For that reason, from quorum values over 20, the time execution decreases faster than the tendency for quorums under 20.

We can explain the flat regions in the Figure 6-14 because the number of occurrences of motifs follows a binomial probability and the time is increasing because there are more motifs to look for.

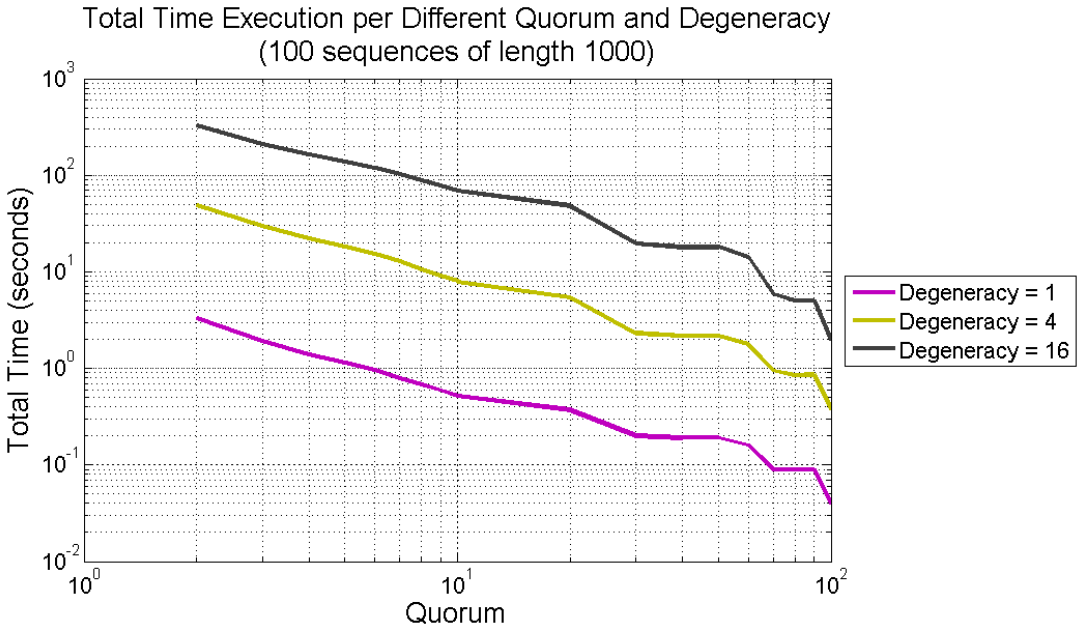

Figure 6-14 - Total Time execution for different Quorum and degeneracy

The quorum and degeneracy values also has an impact in how fast the motif discovery algorithm finds motifs. From a degeneracy point of view we can determine that when degeneracy is bigger the algorithm takes more time iterating through the extra character.

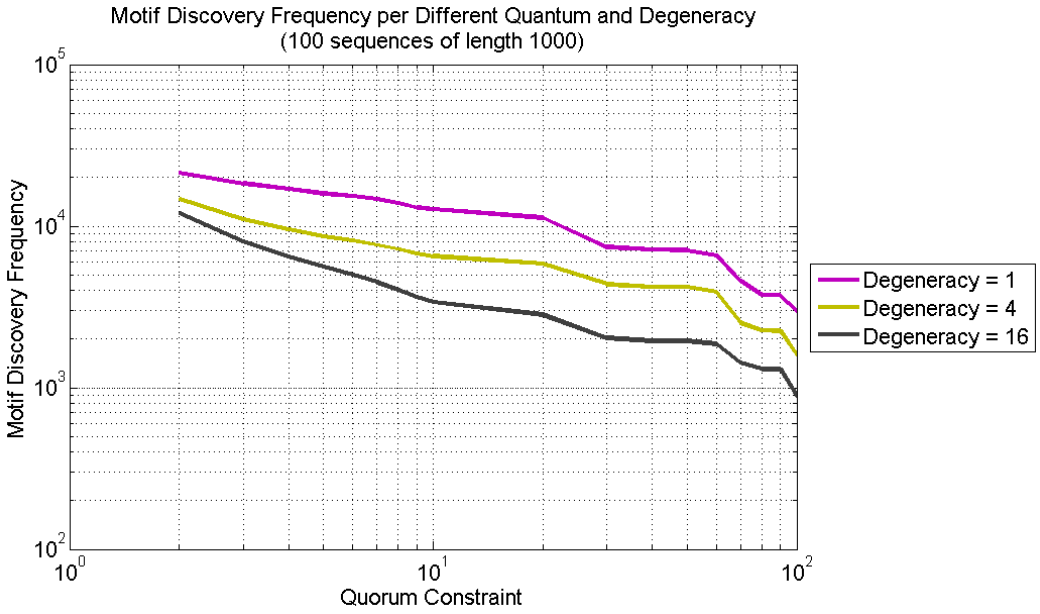

Figure 6-15 - Number of motifs discovered by second per different quorum and degeneracy

The next plot shows the number of motifs sorted by their exact length and for different values of degeneracy, and quorum = 2. Since we are using random data generation, the number of motifs found by the algorithm are subjected only to probabilistic and combinational reasons. As we expected, for short motifs the number of motifs is similar, and while we increase the size, more combinations are possible. We realized the simulation for 100 sequences of length 1000, and for combinational reasons the number of motifs for lengths between 2 and 8 are the maximum number of motifs. But when the length is bigger than 8, the trend starts to change.

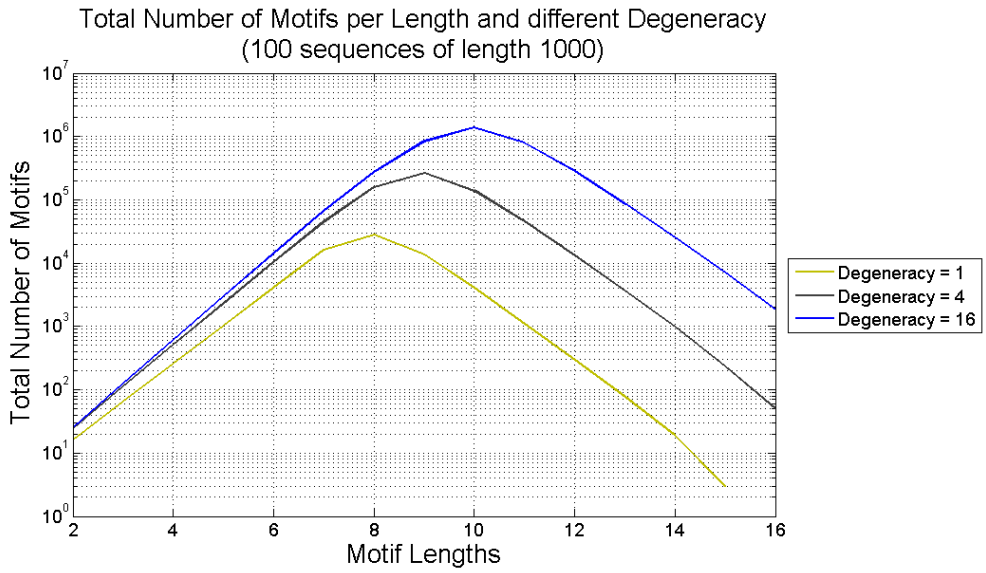

Figure 6-16 - Total number of motifs per Length and Degeneracy

An important attribute of a motif discovery algorithm is the motif discovery frequency, which is the number of motifs found by second. We present in Figures 6-17 and 6-18 two different plots which show the motif discovery frequency looking for motifs with an exact length, or looking for all those motifs with a maximum length *k*.

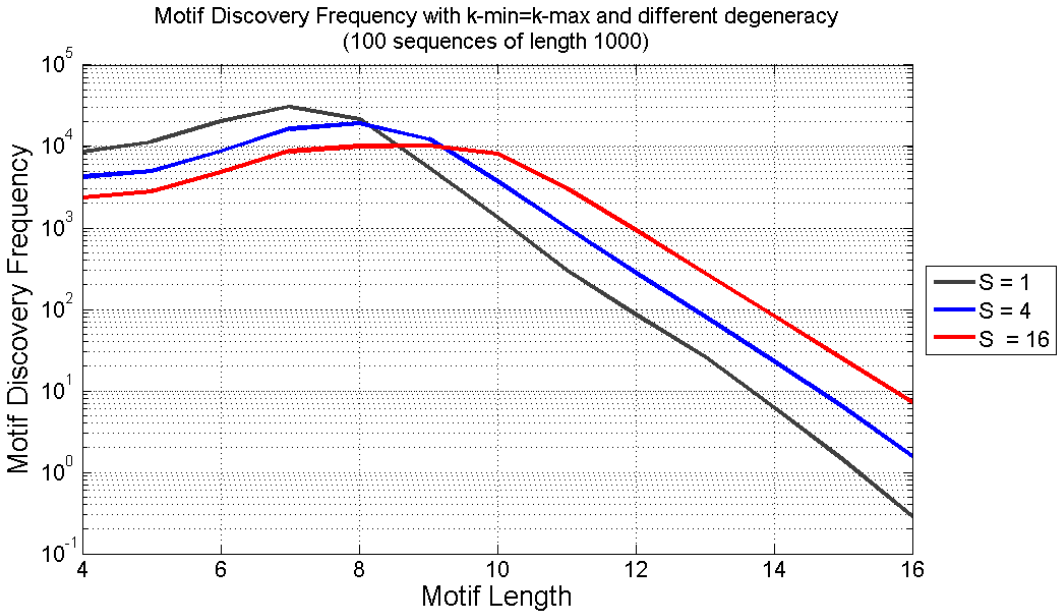

Figure 6-17 - Motif Discovery Frequency per different and distinct motif lengths and different degeneracy, using 100 sequences of length 1000.

In Figure 6-17 we can observe how the frequency for short motifs is bigger when *S*=1, and when reaches a peak around length=7, it changes its behavior and the frequency

drops fast, because we the overhead caused by searching long motifs is bigger than for the short ones, and also because due the binomial distribution of the motifs, at a certain point the possibility that certain motif occurs is lower.

For bigger degeneracies, the initial frequency is smaller because the overhead in the algorithm is bigger since we have to process the degenerate value and move through different paths at the same time. But the frequency remains higher for longer motifs due the introduction of the degenerate character.

Figure 6-18 calculates the motif discovery frequency looking for all the motifs with length shorter than *k_max*. It means that the algorithm will find per each level the valid motifs until the last depth *k* and will add the motifs with length *k*+1.

This condition in combination with the branch and bound algorithm we are using explains why starting from *k_max*=10 the lines remains stable. At a certain point, no more motifs with the requested length exist, and the algorithm stops the execution. Thanks to the accumulated motifs smaller lengths, the frequency remains invariable for the rest of bigger *k_max*.

The efficiency loss after depths=7, 8 and 10 for degeneracy=1, 4 and 16 respectively, is produced because at these levels the trie becomes sparse and the overhead reduces the motif discovery frequency.

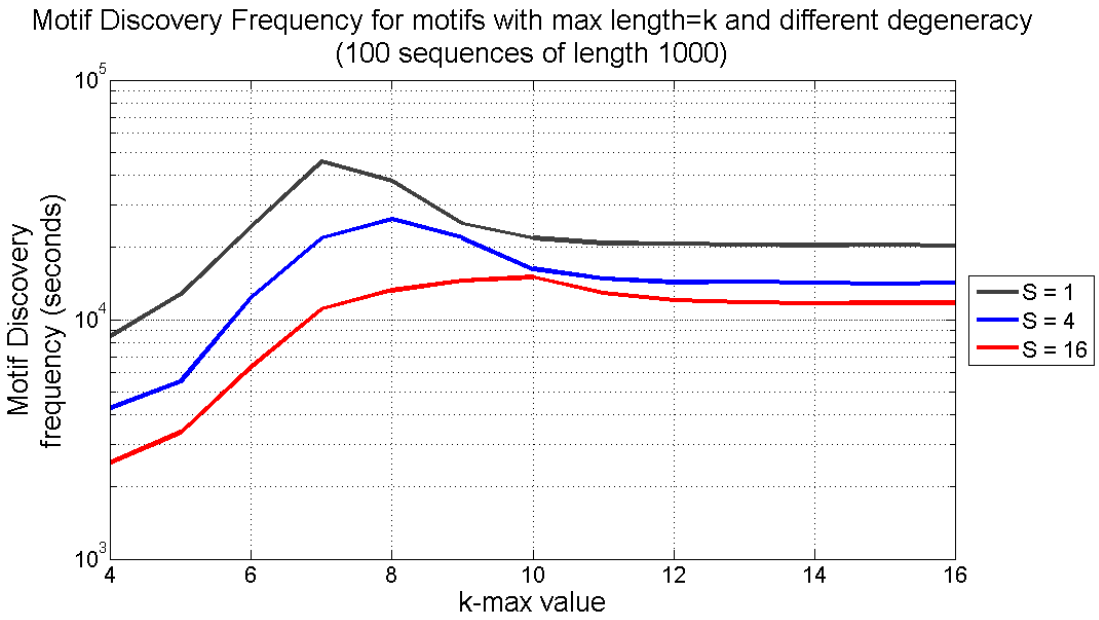

Figure 6-18 - Motif Discovery frequency of motifs with maximum length=k, for different degeneracy and using 100 sequences of length 1000.

6.4.1. Comparison between Suffix Trie and Sliding Window approaches

In Figure 6-19 we can observe how important is the utilization of the branch and bound strategy.

We can determine that for approaches based in exhaustive algorithms, the performance will be always bad when for large data sources and setting some conditions, will be worse than for approaches which can decide if stop processing the non-promising paths.

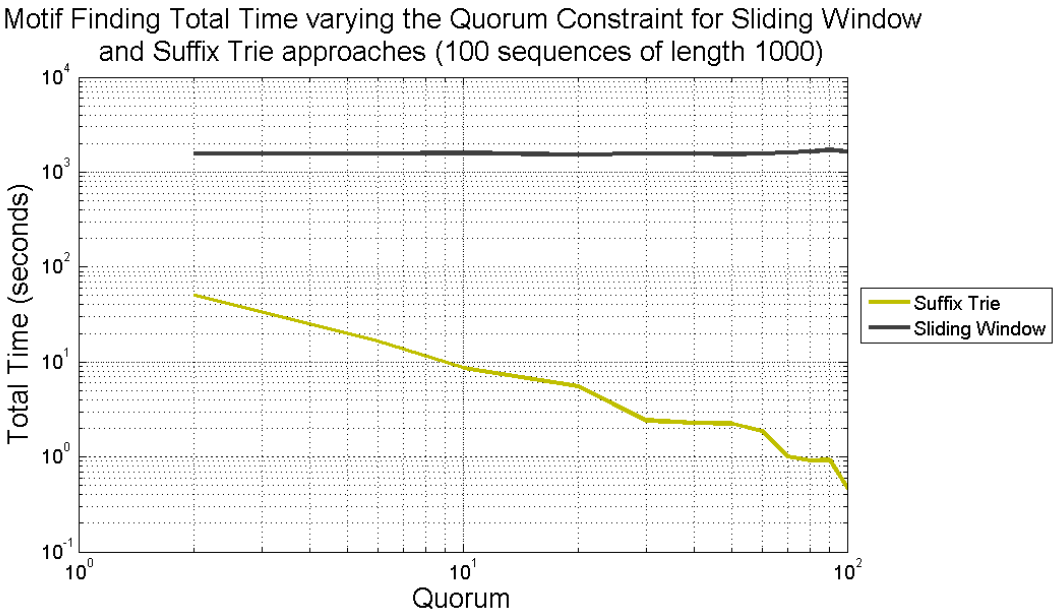

Figure 6-19 - Motif finding total time comparing sliding window approach with suffix trie

Thanks to the utilization of the Suffix Trie structure and the branch and bound strategy, when the size of the problem increases the time execution used by the Suffix Trie approach is much better than with the Sliding Window approach.

We can observe how the time spent in executions increases faster for the sliding window than for the suffix trie. It determines that increasing the sequence length, worse will be the performance of the sliding window compared to the suffix trie.

Sliding Window is almost quadratic, while for the suffix trie is almost linear.

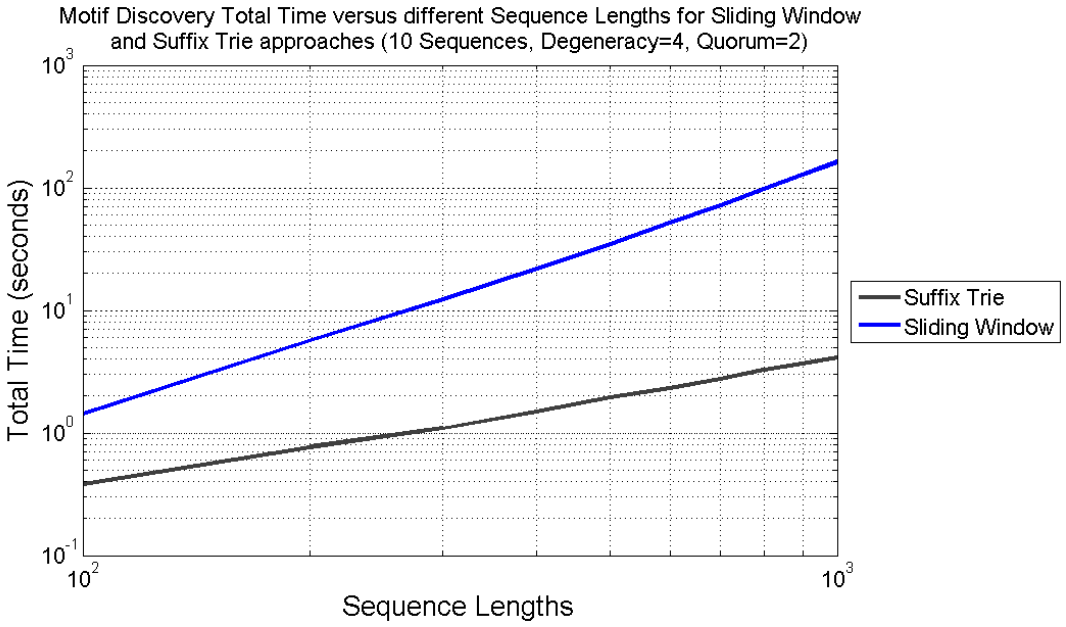

Figure 6-20 - Motif discovery total time comparing Sliding Window Approach and Suffix Trie for different sequence lengths (10 sequences, degeneracy=4 and quorum=2)

In the Figure 6-21 we present the comparison between the motif discovery frequency with Sliding window approach and Suffix Trie. We observe a huge decrease in the motif discovery frequency for the sliding window, while for the Suffix Trie it remains constant, thanks to the branch and bound.

In the Sliding Window the running time is independent of the quorum and we pay a big overhead iterating through all the sequences and motifs, because this exhaustive algorithm has to check all the possible motifs in the sequences to determine if it a valid motif due to the quorum constraint.

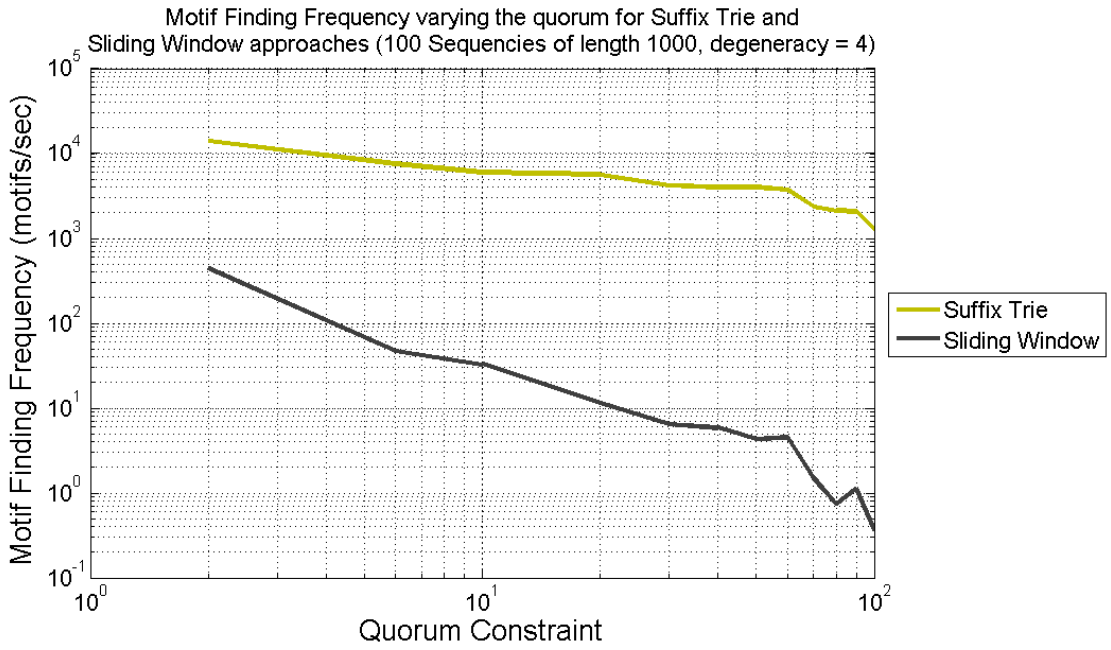

Figure 6-21 - Motif Discovery frequency comparing Sliding Window Approach and Suffix Trie varying the quorum, for degeneracy = 4 (100 sequences of length 1000)

7. Finding the optimal motif

In the last chapter we explained how both exact and degenerate motif algorithms find in a set of sequences all the motifs which satisfy some conditions, like the length or the number of sequences in which a motif must appear.

Now we will explain how to select the optimal motif among all candidates.

After an execution of the motif discovery algorithm one set is returned, and it can contain a huge number of motifs if a small value for the quorum constraint or we are looking for short motifs were set.

Our goal is the optimal motif for each different group which are divided by length and degeneracy.

We calculate first the probability for each character of the alphabet to appear in the sequences.

We iterate for all the sequences counting how many times each character appears in the sequences, and with the total length of all the sequences we can easily calculate the probability per each character.

We start sorting the motifs by their length. Starting for the longest length allowed, we store all those motifs with the same length requested in the same set. We will iterate until all the motifs are stored into the group of motifs with their same length.

At this point we will have as many sets as different motif lengths exist, but probably those motifs have different degeneracy. We will split again the sets into other sets that will contain now the motifs with the same length and with the same degeneracy.

Finally, after sorting by length and degeneracy, we will select the optimal motif for each of the groups calculating the probability that the current motifs appear in the sequences.

We will use the probability per character calculated at the beginning, and we will multiply the probability of each character by the probabilities for the rest of the characters in the motif.

The optimal motif of each group will be the one with less probability of appearing at the sequences.

8. Conclusions

In this thesis we have designed a Suffix Trie to process DNA sequences in order to find an optimal motif.

In the first stages we studied how to design a structure with low memory cost (16 bytes per character versus 44 bytes in the original). We represented our sequences using a Suffix Trie with a limited depth which gave us a fast structure to find patterns.

We used two additional structures to keep the information of the suffixes and provide a fast way to search sequences IDs and suffixes in the trie.

In a second stage we focused on writing a Motif Discovery algorithm which takes into account different restrictions as motif length, degeneracy or quorum.

We did extensive benchmarking and estimated time and memory requirements for different use cases in comparative motif discovery. We compared the memory efficiency of the Suffix Trie and Suffix Tree finding out that the Suffix Trie requires less memory when the trie is dense or the *cutoff depth* is small, but for sparse tries the Suffix Tree has a better memory efficiency.

We analyzed the memory difference between a setup with a trie per gene family or 1 trie to store all gene families at once, concluding that in general the use of a unique trie requires a huge amount of memory compared to creating F different tries. But building the unique trie is faster than building the F tries.

The motif discovery frequency we achieved is around 20.000 and 25.000 motifs per second, being over 30.000 motifs per second when the optimal conditions where set, like motifs of length 8, degeneracy = 1 and large data sets.

Finally we have written a simple solution to select the optimal motif among all the different sequences using a scoring approach, depending on length, degeneracy, quorum and probability of the characters which composes the word.

9. References

[1] http://learn.genetics.utah.edu/content/begin/dna/transcribe/

[2] Degeneracy, Redundancy & Complexity in Biological Systems & Their Measures, Qing-jun Wang

[3] A survey of DNA motif finding algorithms, Modan K Das, Ho-Kwok Dai

[4] http://www.dna.affrc.go.jp/misc/MPsrch/InfoIUPAC.html

[5] An introduction to Bioinformatics algorithms, Neil C. *Jones*, Pavel A. *Pevzner.*

[6] Finding Motifs based on Suffix Trie, F. Zare-Mirakabad, P. Davoodi, H. Ahrabian, A. Nowzari-Dalini, M. Sadeghi, B. Goliaei

[7] A survey of DNA motif finding algorithms, Modan K Das, Ho-Kwok Dai

[8] Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification, M. Sagot

[9] Identification of Consensus Patterns in Unaligned DNA and Protein Sequences: a Large-Deviation Statistical Basis for Penalizing Gaps, G. Stormo, G. Hertz

[10] CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice, Thompson JD, Higgins DG, Gibson TJ

[11] DIALIGN: multiple DNA and protein sequence alignment at BiBiServ, Burkhard Morgenstern

[12] Efficient Implementation of Lazy Suffix, R. Giegerich, S. Kurtz, J. Stoye