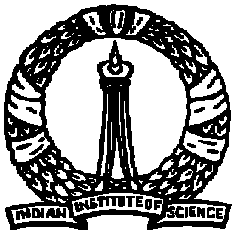
Performance Analysis of Horizontally Compacted String Indexes

A Project Report
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Internet Science and Engineering

by

ANOOP JAIN



COMPUTER SCIENCE AND AUTOMATION
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012

JULY 2004

to *My Parents & Richa*

Acknowledgments

I take this opportunity to express my sincere gratitude towards my project guide Prof. Jayant Haritsa for his guidance, invaluable suggestions and constant support. It would have never been possible for this project to see the light of the day without his help and encouragement.

I also wish to thank whole heartily all the faculty members at Department of Computer Science and Automation for the invaluable knowledge they imparted to me and teaching me a new approach to take up a challenge. I like to extend my thanks to all the staff at CSA department for all the help, direct or indirect.

I am also thankful to fellow DSL'ites Abhijit, Amol, Kumaran, Maya, Parag, Shipra, Srikanta and Suresha for their support and wonderful company in DSL. I wish to thank Amarnath, Girish, Niranjan and Sandhya for their encouragement and moral support. I am grateful to all my friends at IISc who made my stay at IISc really enjoyable and unforgettable. The memories of these two years spent with them would be cherishable for my lifetime.

Last but not the least, I would like to thank my parents and my sister without whom it was not possible for me to reach this stage.

Abstract

Suffix tree is the most extensively studied index structure for longer strings (e.g. genomic strings). On the other hand, DAWGs and CDAWGs are index structures which were developed almost a decade after the suffix trees and support almost all the applications of suffix trees. But these could not get attention of research community much because of its size for DAWG and slow construction procedure for CDAWG. Recently a linear time online direct construction algorithm has been suggested for CDAWG which gives an opportunity to explore the structure. Another index structure *Spine*, recently been proposed, shown to be better than suffix trees.

In the present work, we do an comparative analysis of both Spine and CDAWG for various genomic strings. Implementation of CDAWG is done and algorithmic claims are verified. A comprehensive set of experiments are performed to evaluate performance of both Spine and CDAWG. Results show that in in-memory environments, CDAWG searches are about 2 to 3 times faster than that of Spine, though memory usage of Spine are approximately 40% less than that of CDAWG. Efficient extension of these indexes has been implemented for larger alphabet. The results reveal the generality of these two structures. Finally, indexing has been integrated with BLAST, the popular genomic alignment tool, to study its effectiveness over the heuristic based approximate search.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

String indexing is always considered as one of the favorite topics in research community due to wide variety of its applications. One such example can be *computational biology*, where large DNA sequences are considered as nothing but long strings over alphabet {a,c,g,t} and they are object of linguistic and statistic analysis. As normal string comparison methods do not scale well with such long strings, need of string indexes arises.

Historically, string indexing begins with suffix trie [17], which holds all the suffixes of the data string. *Suffix trees* which can be obtained by merging unary nodes of a suffix trie to its parent node, were suggested as an improvement over suffix tries and as the linear time and space construction algorithms were devised for suffix trees, suffix trees became the most extensively used structure in the present context.

While suffix trees are result of vertical compaction of the suffix tries, yet another structure namely Directed Acyclic Word Graph(DAWG) [2], though not as popular as suffix trees, was introduced later which can be obtained by minimizing the suffix trie. Further, as an improvement over DAWG, Compact Directed Acyclic Word Graph(CDAWG) was presented in [3] which can be obtained from the DAWG in a similar manner as suffix tree can be obtained from suffix trie. Recently, *Spine*, which is a complete horizontal compaction of a suffix trie has been suggested in [12]. Spine is structure wise much more closer to DAWGs.

In parallel to above mentioned works, other approaches like the one used in BLAST [15], the popular genomic alignment tool, were also in progress. This approach does not use any index and is a heuristic search method that seeks words of length W that score at least T when aligned with the query and scored with a substitution matrix. Words in the database that score T or greater are extended in both directions in an attempt to find a local optimal.(See [15] for details). As the approach being heuristic based, BLAST can miss few matches. These matches can be interesting in some cases, especially if the aligned sequences are highly different. This motivates us to analyze the performance of BLAST integrated with an index.

Although few linear time and space construction algorithm are suggested for direct construction of CDAWG, but its performance was not analyzed to the best of our knowledge. In the present work we evaluate the performance of CDAWGs through a comprehensive set of experiments, both in terms of time and space and compare it with latest index structure "Spine".

In addition to that both indexes have been extended to support English text over an alphabet size of 26 and statistical properties have been verified to be same as that of genomic sequences.

1.1 Contributions

The main contributions of the present work can be summarised as follows:

1. The Ukkonen's algorithm based CDAWG construction algorithm [8] has been implemented for the first time and the fact are verified experimentally as well.
2. A few optimizations are suggested for CDAWG to improve its memory usage.
3. We profile the performance of both Spine and CDAWG over various genetic strings for both in-memory and disk based scenarios. Various features and limitations of both the structures are examined.
4. Performance for larger alphabets ($|\Sigma| = 26$) has also been analyzed.
5. Indexes have been integrated with BLAST and performance of BLAST with and without Spine index has been observed.

1.2 Outline of the report

The remainder of this report is organized as follows : Related work is overviewed in Chapter 2. Chapter 3 describes the Spine index structure. CDAWG is explained in Chapter 4. The details of implementation are discussed in Chapter 5. Experimental results are analyzed in Chapter 6. Extension for English text is presented in Chapter 7. Details of integration of Spine with BLAST are given in Chapter 8. Finally we conclude in Chapter 9.

Chapter 2

RELATED WORK

The horizontal compaction of string indexes is one of the topics which is not discussed much in the literature. Historically, Blumer et al. [2] introduced Directed Acyclic Word Graph with a linear time construction. But the restriction with the structure was its size which is around 35 bytes per character for genomic strings. In a subsequent paper Blumer et al. [3] came up with a compacted version of DAWG, namely CDAWG which can also be seen as minimization of Suffix Tree. They also describe a linear time algorithm to obtain it from corresponding DAWG. This reduced the cost of indexing from 35 bytes to around 21 bytes per character of genomic strings. Still, as the construction algorithm requires a DAWG to construct a CDAWG, hence the initial phase requires same amount of memory as of DAWG construct and hence compactness of CDAWGs could not be utilized because of the limitation of construction algorithm.

The first linear direct construction of CDAWG was presented by Crochemore et al. in [4]. The algorithm was based on the McCreight's algorithm for suffix tree construction. It saves about half the space required for ordinary DAWGs or Suffix Tree. Also, as it reduces the number of nodes (about $\frac{2}{3}$ less) and edges (about $\frac{1}{2}$ times less), it reduces the search and construction times as well. The only limitation of the algorithm was that it was not an online algorithm and requires to know the string to be indexed before hand. This limitation was overcome by Inenaga et al. [8] when they provided an *online* linear construction algorithm for direct construction of CDAWGs. This algorithm was based on Ukkonen's algorithm for suffix tree construction.

Full horizontal compression was achieved by Spine which was suggested by Neelapala et al. [12]. The structure has some remarkable properties which includes online direct and linear construction algorithm. It also supports a simple buffering strategy to improve the performance when disk comes into picture. None of the above mentioned structures talk about disk at all.

In contrast to horizontal compression, a lot of work has been done on vertical compression of the trie i.e. suffix trees. There are basically two classes of approaches, first where people tried to reduce the size of suffix trees while other class comprises of approaches which changed the structure to improve its

performance. Kurtz [9] reduced the size of suffix tree to 12.5 bytes per character but the structure was slower with respect to construction and search times due to increased number of edge traversal. Another significant attempt to reduce the size of the suffix tree was Lazy Suffix Tree [5] which further lower down the memory requirements to 8.5 bytes per character, but structure does not have suffix links, hence lacks in functionality.

Suffix arrays [10], which are not suffix trie based, are supposed to be smallest index structure with approximately 6 bytes per character. Although suffix arrays can be constructed in linear time, the search time for suffix array becomes $O(|Query| + log(|x|))$.

For disk based construction of suffix trees, Hunt et al. [7] proposed an phase wise construction algorithm but have a quadratic running time and provides no suffix links.

Other than improving over the structures and suggesting new structures, there is another approach to improve performance of string indexes by adjusting the systems to support the index. One such attempt is presented in [1] by Srikanta et al. They suggest a new simple and static buffering policy, namely Top-Q, to improve the disk based construction of suffix trees, without affecting the suffix tree structure.

Chapter 3

SPINE

Parts of this chapter are mostly based on [12]. Spine is one of the recently proposed string index structure [12]. To the best of our knowledge, it is the only horizontally compacted trie index structure. Although CDAWG comes very close to Spine, still it is minimization of suffix tree or vertically compacted version of DAWG, see Figure 4.2.

The most important property of Spine is the compaction it can achieve which is the extreme of horizontal compaction i.e. a single linear structure. The linearity of the structure and one to one correspondence of nodes to characters in sequence make it distinct from other index structures. Again, linearity results in simple static buffering policy which is a key to improve performance on disk.

3.1 Spine index structure

The central component of SPINE is the "backbone" of nodes connected by forward (or downstream) directed edges called "vertebras", as shown in Figure 3.1. Each vertebra corresponds to a character in the input data string, and this character is used to provide a *character label* (CL) for the vertebra. The vertebras appear in the same order as the associated characters in the input string.

While the backbone forms one source of forward connectivity between the nodes, there are additional downstream edges that connect nodes across the backbone. These edges are called "ribs" (full lines in Figure 3.1) and "extribs" (dotted lines in Figure 3.1). Similar to vertebras, each rib is labeled with a character label (CL), corresponding to the character that it represents in the associated suffix. The set of forward edges collectively represent all possible suffixes of the data string, and are used during the search process.

The backward (or upstream) edges, called "links" (dashed lines in Figure 3.1) are created and used during the SPINE construction process. They provide the ability to process suffixes on a set basis.

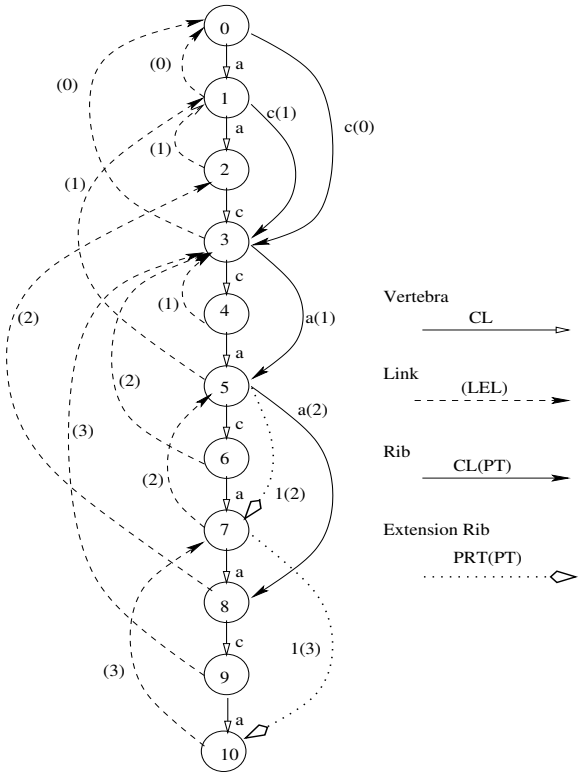


Figure 3.1: Spine structure for aaccacaaca

3.2 Avoiding False Positives

As mentioned earlier, the Spine index represents the complete horizontal compaction of all the suffixes in the corresponding trie. An implication of merging of all the matching paths into a single path is that all paths that were there in the original trie continue to be represented in SPINE, and therefore there is no possibility of *false negatives*. However, *false positives*, that is, invalid substrings, may arise. For instance, in Figure 3.1, a path for accaa appears to exist in the SPINE index even though it is not a substring of the data string.

To avoid such false positives, we take recourse to a numeric labeling strategy for the edges during the construction process. Specifically, each rib and extrib is assigned an integer label, called *Pathlength Threshold (PT)*. The extribs have an additional integer label called *Parent Rib Threshold (PRT)*. In order to be able to assign the correct PT values to the ribs/extribs, each link is assigned an integer label called *Longest Early-Terminating suffix Length (LEL)*. For example, in Figure 3.1, the rib from Node 3 has a PT of 1, the extrib from Node 5 to Node 7 has a PRT of 1 and PT of 2, while the link from Node 8 to Node 2 has an LEL of 2.

These labels, which are assigned during the index construction process as explained later, determine

when forward edges can be traversed during the subsequent search process. Specifically, a rib/extrib can be traversed only if the length of the path traversed so far (i.e. from the root node till that point) is less than or equal to the PT of the rib/extrib. So, for example, the accaa path will not be permitted in Figure 3.1. This is because when we traverse the path from the root for acca, after we reach Node 5, the rib for a violates the constraint because it has a PT of 2, which is less than the current pathlength of 4. Thus, accaa is not a valid substring of the given data string.

Overall, a search path in a SPINE index is a *valid path* if and only if all the ribs/extribs in the traversed path satisfy the PT constraints.

3.3 Notation and Terminology

In the remainder of this section, we describe the components of SPINE in detail. Our discussion assumes that the data string which is being indexed is composed of M characters. For ease of presentation, we use the notation shown in Table 3.1. While the table entries are mostly self-explanatory, the *termination* concept requires elucidation: A suffix s_{ij} is said to terminate at node p ($p \leq i$) if there is a valid traversal path from the root node to node p whose string of character labels match the suffix. A suffix s_{ij} whose termination node is strictly less than i is said to be an *early-terminating* suffix, otherwise it is called *end-terminating*.

To make the above notation clear, consider Node 5 in Figure 3.1, for which S_5 = aacca, s_{52} = ca, $AllSuf_5$ = {aacca, acca, cca, ca, a}, $EndSuf_5$ = {aacca, acca, cca, ca}, $EarlySuf_5$ = {a}.

Notation	Meaning
N_i	*Node i*
S_i	*String on backbone from root to* N_i
s_{ij}	*Suffix of* S_i *of length* j
$AllSuf_i$	*Set of all suffixes of* S_i
$AllSuf_i(k)$	*Set of suffixes of* S_i *of length* $\leq k$
$EndSuf_i$	*Set of suffixes in* $AllSuf_i$ *terminating at* N_i
$EndSuf_i(k)$	*Set of suffixes of* $EndSuf_i$ *of length* $\leq k$
$EarlySuf_i$	*Set of suffixes in* $AllSuf_i$ *not terminating at* N_i
$Link(N_i).LEL$	*LEL of the link of* N_i
$Link(N_i).Dest$	*Destination node of link of* N_i
$Rib(N_i).Dest(c)$	*Destination node of rib at* N_i *for character c*
$Rib(N_i).PT(c)$	*PT of rib at* N_i *for character c*

Table 3.1: Notation

3.4 Vertebra Backbone

During construction, the backbone is initially created with a single node, called the *root node*, and for each character in the data string, a new node is added sequentially using a vertebra edge labeled with the corresponding character. The node that is currently at the bottom of the backbone is referred to as the *tail node*, N_{tail}. Each node has an integer identifier which is set equal to the length of the backbone string above that node. With this naming convention, the root node has identifier 0, the first node has identifier 1 and so on until the tail node of the entire index which will have identifier M.

3.5 Links

Links are meant to record, at each node, the information about its early-terminating suffixes, namely, $EarlySuf_i$.

Specifically, only the *longest* early-terminating suffix (hereafter referred to as LET-suffix) is explicitly kept track of since, by definition, all shorter suffixes are also early-terminating suffixes and they would themselves have been linked up earlier. For example, in Figure 3.1, there is a link from N_5 to N_1 to represent a, the LET-suffix. If a node has no early-terminating suffixes (i.e. $EndSuf_i = AllSuf_i$), then its link points to the root node, N_0, which can be interpreted as representing the null suffix. N_3 in Figure 3.1 is an example of this scenario. Finally, as a special case, the root node has no link edge since it is the starting node.

Link Labels

The LEL label of a link is the length of the LET suffix which it represents. Intuitively, if we have a link from N_i to N_j with a LEL 'k', then it means $s_{ik} = s_{jk}$. More formally, $AllSuf_i$ can be defined as follows:

$AllSuf_i = EndSuf_i \cup EarlySuf_i$ and

$EarlySuf_i = AllSuf_j(k)$

where $k = Link(N_i).LEL$ and $j = Link(N_i).Dest$.

3.6 Ribs

When the SPINE index that has been built for S_i is extended by one more character from the data string, all the suffixes of S_i need to be extended by this additional character, c_{tail}. For the end-terminating suffixes, the newly added node on the backbone, N_{tail}, *automatically* records this extension through its vertebra edge. For the early-terminating suffixes, however, the extension must be explicitly recorded and this is achieved through the addition of rib edges. Specifically, the link chain from N_i is traversed and if

a rib/vertebra does not already exist for c_{tail} at any node, say N_j, in the link chain, a new rib is created from that node to N_{tail}.

The traversal of the link chain terminates if either the root node is reached, or a node having an outgoing edge labeled with c_{tail} is reached. The first stopping condition is obvious since no further traversal is possible, while the other condition reflects the fact that the suffix in question has *already* been previously extended. And there is no need to explicitly handle the remaining smaller suffixes as they would also have been extended automatically.

Rib Labels

When a new rib is created at N_j, its CL is set to c_{tail} and its Pathlength Threshold (PT) is set to the length of the longest suffix of S_i terminating at that node, which is given by the LEL of the last traversed suffix link. Intuitively, the rib PT represents the length of the longest prefix that can be traversed from the root before the rib is traversed. This is because the rib was created to extend the suffix of that length. This means that, as mentioned earlier, a rib at N_j can be traversed during the SPINE search process only if the length of the current traversal path is $\leq$ PT of this rib.

3.7 ExtRibs

As mentioned above, the link-chain traversal terminates for rib addition if the current node already has a matching rib (i.e. with CL = c_{tail}). However, the following situation may now arise: The PT of the pre-existing rib may be *less* than the LEL of the link used to reach this node, which means that this rib is not valid to represent the extension of the associated early-terminating suffix. To address this issue, the solution that immediately comes to mind is to update the rib's PT to be equal to the LEL value. However, this is not correct since it may permit illegal paths resulting in false positives. An alternative approach has been taken for extending the rib itself through edges called *extribs* (extension ribs). For example, in Figure 3.1, the extrib (dotted line) from N_5 to N_7 is an extension of the "parent" rib connecting N_3 to N_5.

At a given node, there may be multiple extribs, each corresponding to a different parent rib that terminates at this node. From an implementation perspective, this is problematic since it makes the node size to be variable. Therefore, the alternative approach of maintaining the extribs in a *chained* fashion has been proposed. That is, the first extrib in the chain is located at the destination node of the rib which failed the pathlength threshold test, and the second extrib is located at the destination node of the first extrib, and so on. This ensures that at any node there is at most *only one extrib*. So, whenever an extrib is need to be created, instead of creating it from the destination of the parent rib, traversal to the node at the end of the extrib chain is done, and then a new extrib is created from this node to the tail node. All the extribs created for a rib are its children.

ExtRib Labels

Each extrib has an associated Pathlength Threshold (PT), which is the length of the longest suffix that it is extending, as well as a PRT, which is the PT value of the parent rib. The reason for including the PRT value is to be able to uniquely identify the extrib. Note that a character label is not required for an extrib as it is implicitly represented by the CL of the incoming rib or extrib at its source node. And hence, a complete extrib chain represents a single character. In Figure 3.1, an example chain is the extrib from N_5 to N_7, and then from N_7 to N_{10}.

Chapter 4

COMPACT DIRECTED ACYCLIC WORD GRAPHS

The first ever linear-sized graph to represent the sub-words of a word, called Directed Acyclic Word Graphs was presented in [2]. When terminal states are added to DAWG, it becomes the minimum automaton accepting suffixes of a word.

Later, in [3], Blumer et al. presented *Compact Directed Acyclic Word Graphs* which was a space efficient version of DAWG and can be obtained by deleting all the nodes of out degree one and corresponding edges. They also presented a linear time algorithm to obtain CDAWG from corresponding DAWG.

Directed Acyclic Word Graph [4]: A suffix automaton of a word x, denoted as $DAWG(x)$, is the minimal deterministic automaton (not necessarily complete) that accepts $S(x)$, the (finite) set of suffixes of x.

Before defining CDAWG, let us define syntactic congruence relation. The syntactic congruence associated with $S(x)$ is denoted by $\equiv_{S(x)}$ and is defined, for $x, u, v \in \Sigma^*$, by:

$$u \equiv_{S(x)} v \iff u^{-1}S(x) = v^{-1}S(x)$$

That is, u and v occur as prefixes of the same suffixes of s. In other words, the occurrences of u and v must end at same position in the string. Hence, if u and v occur in the string, one must be a suffix of other. We call classes of factors the congruence classes of the relation $\equiv_{S(x)}$. The longest word of a class of factor is called the representative of the class. States of DAWG(x) are exactly the classes of the relation $\equiv_{S(x)}$. The class of all strings which are not substrings of x is called *degenerate* class. The longest string in a *non-degenerate* class of factors is the *representative* of the class.

Compact Directed Acyclic Word Graph [4]: The Compacted Directed Acyclic Word Graph of a word x, denoted by $CDAWG(x)$, is the compaction of $DAWG(x)$ obtained by keeping only states that are either terminal states or strict classes of factors according to $\equiv_{S(x)}$, and by labeling transitions accordingly.

4.1 CDAWG Index Structure

The point to mention about the CDAWG is its simple structure. CDAWG structure comprises of nodes along with its forward edges and suffix link. Figure 4.1 shows an example graph for $CDAWG_{aaccacaaca}$. The solid arrows represent forward edges and broken arrows are the suffix links. Each forward edge has a multi letter transition associated with it. No two edge label can begin with the same letter on a single node. Each node in CDAWG structure represents a strict class of factors. Each state has at least two or at most $|\Sigma|$ outgoing edges (which is obvious from definition). All the states other than Initial and Final have suffix links(Section 4.1.2).

4.1.1 Forward Edges

Forward edges are main traversal path during search. Each state can have as much as $|\Sigma|$ outgoing edges and as low as two edges. There is no limit on the number of incoming edges. Each edge is defined as a transition (p, α, q) where p is the source state, q is the target state and α is the associated label. Every path formed by forward edges is a valid path. The string obtained by concatenation of the labels of edges from root node to any state (or at any intermediate point on an edge) will be a factor of the indexed sequence. Also, every factor of the indexed sequence x has a matching path in $CDAWG_x$. In other words, each and every factor of x, and factors of x only, form paths in $CDAWG_x$.

4.1.2 Suffix Links

Definition: Let p be a node of the $CDAWG(x)$, different from the initial or final node. Let β be the representative of the class associated with p. The *Suffix Link* of p, denoted by $Suf(p)$, is the node q whose representative γ is the longest suffix of β whose path does not end at p.

If γ is empty then $Suf(p)$ is the initial node. Suffix links are not defined for initial and final nodes. Although the definition of suffix link does not guarantee that every node in the graph has a suffix link, we have following property:
Any node created during phase i+1 will have a suffix link from it by the end of phase. Here phase is *phase* in construction procedure discussed later in Section 4.5.

4.2 Size Bounds

Number of states: Given $x \in \Sigma^*$, if $|x| = 0$, then $States(x) = 1$; if $|x| = 1$, then $States(x) = 2$; otherwise $|x| \geq 2$, and $2 \leq States(x) \leq |x| + 1$.

Maximum number of states results in a situation when x is of the form of $a^{|x|}$, where $a \in \Sigma$ and minimum occurs when we have x is composed of pair wise different letters. (See [4] for proof)

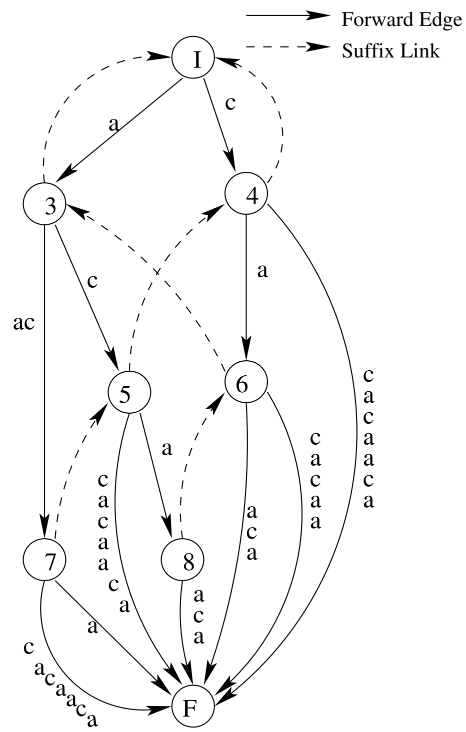


Figure 4.1: CDAWG for aaccacaaca

Number of edges: Given $x \in \Sigma^*$, if $|x| = 0$, then $Edges(x) = 0$; if $|x| = 1$, then $Edges(x) = 1$; otherwise $|x| \geq 2$, and $States(x) \leq 2|x| - 2$.

Maximum number of edges results in a situation when x is of the form of $a^{|x|-1}b$, where $a, b \in \Sigma$. (See [4] for proof)

4.3 Construction of CDAWG

Compact Directed Acyclic Word Graph for a string x can be constructed in both indirect and direct way. Actually $CDAWG(x)$ can be seen as minimization of suffix tree of x or compaction of $DAWG(x)$ as shown in Figure 4.2. Here. we would like to differentiate between minimization and compaction. Though both results in reduction of number of states in the automaton, compaction does not change the basic structure of the automaton and it just merges unary child nodes to its parents. Compaction is just a space efficient representation of an automaton. On the other hand, minimization is a process which changes the structure itself to achieve the minimum number of states so that the language accepted by the automaton should not get changed. How we can construct $CDAWG(x)$ from suffix tree of x is given in [6] while obtaining it from $DAWG(x)$ is discussed in [4].

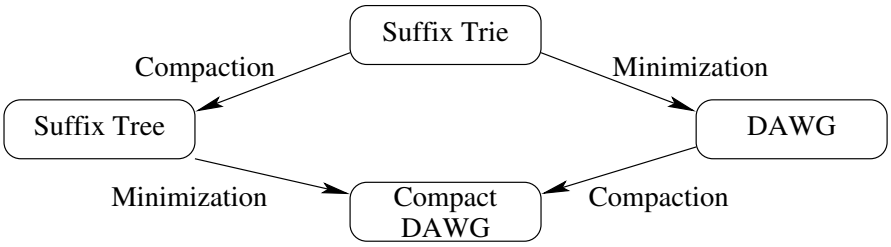


Figure 4.2: Relationships between string indexes

A couple of direct construction algorithms are proposed in [4] and [8]. Former is based on McCreight's algorithm for the construction of suffix tree [11], while the latter is based on Ukkonen's algorithm [13]. The run times of both the algorithms are linear with respect to input string length. In the following text we follow the second approach because of its *online* nature i.e. it processes the characters of input string from left to right one by one, with no need to know the entire string before hand.

4.4 Horizontal Compaction in CDAWG

Though there is no explicit horizontal compaction procedure in CDAWG, it inherits the horizontal compaction from the original DAWG. The horizontal compaction in DAWG is a result of minimization process. For example, in case of suffix trees, once a path gets separated from other, it never meets it again. Hence, tree keeps on growing in width as well. On the other hand, in case of DAWG, if prefixes of a suffix do not lie in the same equivalance class of syntactic congruence relation ($\equiv_{S(x)}$), then those prefixes are represented by two different nodes, but the suffix itself is represented by a single node.

4.5 Description of Construction Algorithm

This section is based on discussion given in [8].

The algorithm is based on Ukkonen's Algorithm for construction of *suffix trees* [13]. Given an alphabet Σ, let $s = s_1, s_2, .., s_n$ be a string on Σ. The algorithm is divided in n phases similar to Ukkonen's algorithm, building at each phase i the implicit $CDAWG$ $\mathcal{G}_i$ for each prefix $s[1..i]$ of s. The implicit $CDAWG$ $\mathcal{G}_{i+1}$ for $s[1..i+1]$ is constructed starting from $CDAWG$ $\mathcal{G}_i$. Each phase $i+1$ is divided in $i+1$ extensions, one for each of the $i+1$ suffixes of $s[1..i+1]$. In extension j of phase $i+1$, the algorithm finds the end of the path from the initial node labeled with substring $s[j..i]$, and extends it by adding character s_{i+1} to the path, unless it is already there. Therefore in phase $i+1$, substring $s[i..i+1]$ is first put on the graph , followed by $s[2..i+1]$, $s[3..i+1]$, and so on. Extension $i+1$ of phase $i+1$ adds the single character s_{i+1} after the initial node. The initial graph $\mathcal{G}_1$ has one initial node I and one

final node F, connected by ab edge labeled by character s_1. The algorithm can be sketched as follows:

1. Construct graph G.
2. **for** $i = 1$ *to* $n-1$ **do**
3. **for** $j = 1$ *to* $i+1$ **do**
4. Find the end of the path from I labeled $s[j..i]$
5. Add character s_{i+1} if needed
6. **done**
7. **done**

At extension j of phase $i+1$, once the end of the path $s[j..i]$ has been located, the $CDAWG$ can be updated according to three different rules:

1. In the current graph, the path $s[j..i]$ ends at F. To update the graph, character s_{i+1} is appended to the label of the edge entering F.

2. The path corresponding to $s[j..i]$ does not continue with s_{i+1}, but continues with at least one character c. If the path ends at a node p, we create a new edge (p, s_{i+1}, F). Otherwise, we create a new node q at the end of the path, splitting the edge in two at the point where the path ends. Then, we create a new edge (q, s_{i+1}, F).

3. Some path at the end of $s[j..i]$ continues with s_{i+1}. In this case, substring $s[j..i+1]$ is already in the current graph: No need to do anything (hence the implicit $CDAWG$).

These rules, however, do not guarantee that at the end of the phase we correctly constructed a $CDAWG$. In fact, the algorithm must also check whether a substring strictly congruent to another one has been encountered, or, conversely, whether a substring has to be removed from a strict class of factors, so that at the end of phase $i+1$ paths ending at the same node correspond to strict classes of factors of $s[i..i+1]$, and vice versa. In the following discussion, let us see hoe the algorithm has been modified. A detailed algorithm in form of pseudo code, is given in the appendix A.

4.5.1 Detecting Strictly Congruent Factors

Two substring α and β belong to the same class C iff they are prefixes of the same suffixes, and there are at least two characters $a, b \in \Sigma$ such that αa, αb, βa, βb are factors of s. Moreover, α must be a suffix of β or vice versa. Let us assume without the loss of generality that $\alpha = c\beta$, with $c \in \Sigma$. We also assume that α and β have occurred just once, that substring αa and βb have been put in the graph in some previous phase (in two consecutive extensions), and in the current extension e have to insert αb. The path spelling α ends in the middle of an edge, and the next character on the edge is a. A new

node p is created at the end of the path, as well a new edge (p, b, F). At the following extension, we have to locate β in the graph. If β has occurred only once (together with α), it now belongs to the same strict class of factors, and we end in the middle of a *non-solid* edge that continues with a. In this case, we *redirect* the edge to p, labeling it with the part of the label that was contained in the path of β (see Figure 4.3. Since there can be more than two consecutive substrings to be assigned to the same class, it is possible that we again end along non-solid edges in the following extensions. In such case, we redirect the non-solid edges to p as well , until we reach an extension where we end at a node or along a solid edge. Otherwise, if β had previously occurred also by itself, either the path corresponding to β ends at a node (β has been followed by characters different from a), or the edge we end on is solid (β has been followed only by a). In the former case, if there is not an edge labeled b leaving the node we create a new edge labeled b to the final node. In the latter case, we create a new node and connect it to final node with an edge labeled b. Then there may be again non-solid edges that have to be redirected to newly created node.

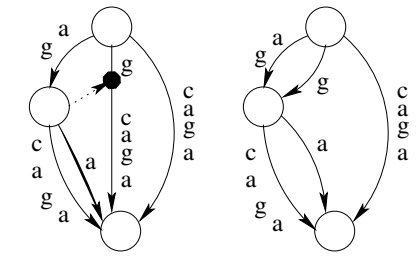


Figure 4.3: Redirecting an edge

4.5.2 Splitting a Strict Class of Factors

Conversely, a substring that has been assigned to a strict class of factors has to be removed from the class if it does not occur as a suffix of the representative when a new character s_{i+1} is added to the string. Let α and β, $\alpha = c\beta$, be the two substrings assigned to the same class in the example of previous section. Now suppose that in phase $i + 1$ we have to insert β in the graph. In this case, s_{i+1} is the last character of β, and we find it at the end of the edge entering node p, that is non-solid, since β is not representative of the class. Now we have two cases: s_{i+1} was found at the end of an edge that entered node p also at the previous extension, or we ended up somewhere else. In the former case, we had also inserted α at the previous extension of the same phase, therefore β still belongs to the same class. In the latter case, we have detected an occurrence of β not preceded by α, that is , not as a suffix of α, and we have to remove it from the class. To reflect this in the graph, we *clone* the node p into a new node q, and redirect the non-solid edge to q keeping the same label. The redirected edge becomes solid (See Figure 4.4. If also

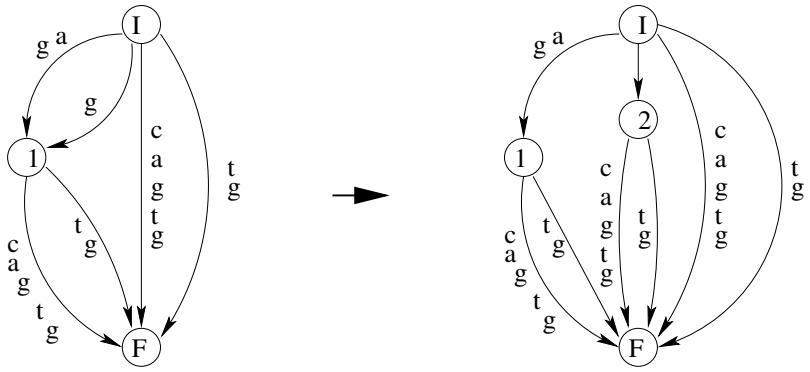


Figure 4.4: Cloning of a state : State 2 is created as clone of state 1

some suffixes of β had been previously assigned to the same class as β, in the following extensions we will again find s_{i+1} at the end of a non-solid edge entering p. These edges are redirected to q. It can be proved that it suffices to check only the last edge on each path to ensure that a class has to be split. No cloning takes place if the character is found at the end of an edge entering the final node.

The observations made in the previous two sections (4.5.1 and 4.5.2) can be implemented in the algorithm by modifying rules 2 and 3 accordingly. Also, it is a point to mention that both redirection of edge to newly created node and node cloning can take place during the same phase.

4.5.3 Using Suffix Links

Naively, locating the end of $s[j..i]$ in the extension j of phase $i+1$ would take $O(i-j)$ time by walking from the initial node and matching the characters of $s[j..i]$ along the edges of the graph. This would lead to an overall ($O(n^3)$ time complexity for the construction of the whole graph. We can reduce it as done in Ukkonen's algorithm for suffix tree construction [13] by using the suffix links which were introduced in Section 4.1.2.

Let us suppose that the algorithm has completed extension j of phase $i+1$. Suffix links are used to speed up the search for the remaining suffixes of $s[j..i]$. Starting from the end of $s[j..i]$ in the graph, we walk backwards along the path corresponding to $s[j..i]$ up to either the initial node or a node p that has suffix link. This requires traversing at most one edge[1]. Let γ be the concatenation of the edge labels of the path from p to $s[j..i]$. If p is not initial node, we move to node $suf(p)$ and follow from it the path spelling γ. Otherwise, we search for $s[j+1..i]$ starting from I. Finally we append s_{i+1} according to one of extension rules, redirecting an edge or cloning a node if needed.

A path spelling γ starting from $suf(p)$ always exists, since all the suffixes of $s[j..i]$ are already in

[1]During any phase, the only node of the graph other than initial and final without a suffix link is the last created one

the graph. Thus, to find the path spelling γ we just match the first characters on the edges encountered. To obtain a linear algorithm, we need two more tricks :

1. When during any extension Rule 3 is applied, that is, a given substring $s[j..i+1]$ is already on the graph, then the same will apply to all further extensions, since all other suffixes of $s[j..i+1]$ are already in the graph as well. Therefore, once Rule 3 applied (and no node has to be cloned or edges redirected), we can stop and move to next phase, since all the strings to be inserted are already in the graph and no adjustment is needed for the classes.

2. If a new edge is created entering the final node during extension j of any phase i, then Rule 1 will always apply at extension j in any successive phase. That is, new characters will always be appended at the end of the last edge in the path associated with $s[j..i]$, that will enter the final node. Thus when new edge is created entering the final node with label $s[j..i+1]$, we label it with integers h and $e(j \leq h \leq i+1)$, where e denotes the current phase, that is, the current end position in the string. If we implement e with a global variable and set it to $i+1$ at beginning of each phase $i+1$, we perform implicitly all the extensions to open edges (edges ending at final node).

Chapter 5

IMPLEMENTATION DETAILS

For the experimental purposes, we developed a prototype version of CDAWG. Implementation is based on as discussed in [4]. CDAWG is basically a compact form of automaton. Transition matrices and adjacency lists are two classical approaches for implementation of automata. The former gives direct access to each and every transition but the memory space requirement is $O(states(x) \times |\Sigma|)$. On the other hand, latter stores only existing transitions but requires a $(O(log(|\Sigma|))$ time to access them using standard search techniques. So, this suggests that for a small alphabet such as genomic strings where transition matrices are not sparse enough, transition matrix based implementation is a better choice. Here, in the following discussion, for the ease of presentation we assume genomic strings are being indexed and hence, we will use a transition matrix based implementation.

Now, we describe exact state and edge structures. We have an alphabet of size four, hence transition matrix will contain four transitions per state. For each outgoing transition, we need a target state which can be encoded using a four byte integer. For a DAWG this much information is enough as each of the transition is just a single character transition. But in case of CDAWG, we also have to keep information about the multi letter property of each transition. Now, as we already discussed, each state of CDAWG

Field Name	size	Count	total size
EDGE			
isSolid	0.128	1	0.128
length	4	1	4
target	4	1	4
STATE			
endpos	4	1	4
slink	4	1	4
EDGE	8.128	4	32.5

Table 5.1: Node Structure Components

represents strict class of factors w.r.t. $\equiv_x^R$ relation, which means all the labels of incoming transitions of a state end at same position in indexed string. Hence, we store a value (named as $endpos(s)$) with every state. Also, in addition to it, each edge has a length field with it. Now, using these two information, label of a transition can be determined. Each edge also contain one boolean field which show whether the edge is a *solid* one or not. Each state also has to keep a suffix link pointer which is necessary to implement a linear time construction algorithm. This can be done using a four byte integer. Table 5.1 shows final layout of the structures. We can see that the final state size is 40.5 bytes.

Chapter 6

EXPERIMENTAL ANALYSIS

A comprehensive set of experiments were conducted to evaluate performance of both CDAWG and Spine. The code for Spine was taken from the authors of [12] and CDAWG was implemented as discussed in Chapter 5 using the algorithm discussed in Chapter 4. Our experiments were conducted on the following genomes.

ECO : E.coli bacterial genome of length 3.5 million characters;

CEL : C.Elegans earthworm genome of length 15.5 million characters;

HC21 : Human chromosome 21 genome of length 28.5 million characters;

HC19 : Human chromosome 19 genome of length 57.5 million characters.

All the experiments were conducted on a Pentium-IV 2.0 GHz PC with 1GB RAM and running Red Hat Linux 8.0 operating system. The performance metrics which comprise not only time but memory space requirement also, are as follows.

Index Construction Time: [1] This is the overall time taken to build the complete index for a string.

Index Search Times: This refers to the time taken to perform the search operation of both prefix search and multiple matches search.

Memory requirement: This is the total memory requirement of the index structure at time of construction as well as search operation.

[1] All time numbers in all the experiments are taken as wall clock time.

CDAWG			
Dataset	#Chars	#Nodes	#Edges
Ecoli	3514912	1904844	5076291
Celegans	15077376	7737091	20476457
HC21	28508160	14724580	38878073
SPINE			
Ecoli	3514912	3514912	3718858
Celegans	15077376	15077376	14484179
HC21	28508160	28508160	27164335

Table 6.1: Number of nodes and edges in CDAWG and Spine

6.1 Structural Properties

Before looking into the performance of the two structures, lets have a look at the structural details of the two. Table 6.1 shows the number of nodes and edges presented for various sequences for both the structures. As already discussed, the number of nodes in Spine structure is always equal to the number of characters present in the indexed sequence. But for CDAWG, the number of nodes is almost equal to half of the length of the index sequence. On the other hand, number of edges in CDAWG is almost 1.5 times of the sequence length and almost equal to sequence length in case of Spine.

Locality of accesses

One interesting property to observe is the locality of the access pattern for any index. Though the forward edges are uniformly distributed in both the structures, the targets of backward edges (Links in Spine and Suffix links in CDAWG) are highly clustered. Exact distribution of targets of backward edges is shown in Figure 6.1. This observation is important for deciding the buffer replacement policy for the index.

6.2 In-Memory Analysis

Construction Time

The performance of Spine and CDAWG with respect to index construction time is shown in Figure 6.2. First thing which attracts the attention is the missing bar for $CDAWG_{HC19}$, which is due to the fact that we could not build the CDAWG index for HC19 *within practical time limits* with the resources used for experiments. But if the resources are enough, as in other cases, we can see that CDAWG construction time is about two third of that of Spine. The reason for the faster construction CDAWG is the faster search rate of CDAWG which is discussed next.

Another point to note is the construction rate of indexes which is about 1.7 sec/MB for Spine and

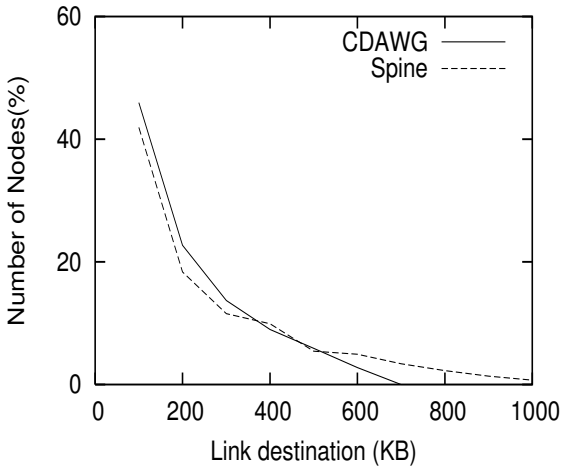


Figure 6.1: Link Distribution for Spine and CDAWG

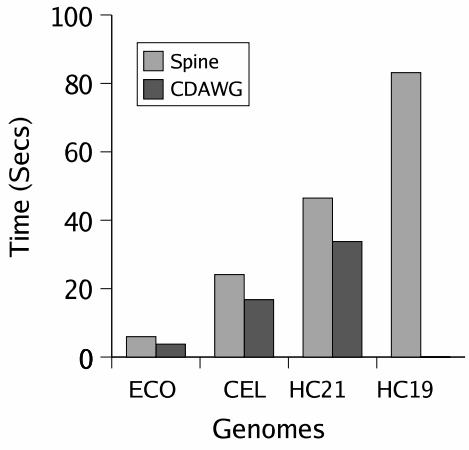


Figure 6.2: Construction time in memory

about 1.3 Sec/MB for CDAWG. Also important point is that construction rate is almost constant within experimental limit which is an experimental confirmation to the fact that both indexes follow a linear time construction algorithm. Also the build rate suggest that with sufficient resources, in-memory construction of human genome will take about $1\frac{1}{2}$ hours.

Indexed Seq.	Query	Spine	CDAWG	Avg mLen
ECO	CEL	8.25	2.41	10.80
ECO	HC21	8.19	2.41	10.58
CEL	ECO	9.36	3.00	11.43
CEL	HC21	9.84	3.18	12.07
HC21	CEL	10.37	3.52	12.57
HC21	HC19	11.35	6.61	16.31
HC19	HC21	13.75	–	15.67

Table 6.2: Average search times(Microseconds) for single prefix search

Search time

Table 6.2 show average search times for single prefix searches averaged over 10000 queries, uniformly chosen from the query string. From Table 6.2, we can see clearly that CDAWG is about 3 times faster in case of single prefix search. The reason seems to be the extra rib/extrib traversals in Spine search process, while in case of CDAWG the traveled path length is always equal to match length. For confirming the same, we looked at the extra rib/extrib traversals with respect to different match-length results for one of the sequence ECO with CEL as query string. The result is shown in Figure 6.3. We can see from the figure that there are almost equal number of extra traversals with respect to match length. It means the same amount of extra comparison operations, because comparison with the vertebra edge has to be performed always. To make it more clear lets see following example.

Example 6.2.1 *Lets consider Figure 3.1 and Figure 4.1 which show Spine and CDAWG indexes for aaccacaaca resp. Now, lets take a query acaaa. When we start traversal in Spine, after following vertebra edge for a from root, we follow rib for c at it has a $PT = 1$ and we reach node 3. Now the PT of the rib labeled a is 1 and current traveled length is 2, hence we can not follow the rib, instead the extrib of the same should be followed. The child extrib has $PT = 2$, which means a valid traversal. Hence, after aca we reach node 7. After it we follow the vertebra edge labeled a and after that we stop as no valid path is present and report it as maximum match with match length of 4. Hence total number of comparisons is 8 out of which 3 are due to rib/extrib traversal.*

On the other hand, in case of CDAWG, path followed is node I, followed by node 3, node 5, node 8 and we stop as soon as we find a mismatch on the edge from node 8 to node F. So in this case, number of comparisons is 5 which is one more than match length.

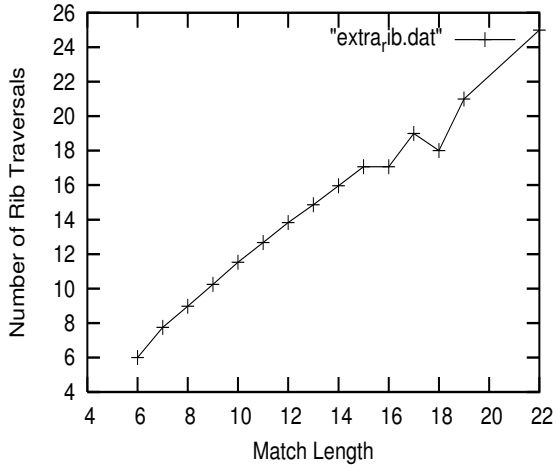


Figure 6.3: Average number of extra rib/extrib traversals in Spine with indexed sequence as ECO

Indexed Seq.	Query Seq.	Spine	CDAWG
ECO	CEL	18.65	10.81
ECO	HC21	35.85	21.91
CEL	ECO	46.95	30.27
CEL	HC21	36.90	24.30
HC19	HC21	29.87	–

Table 6.3: Average search times(Sec) for multiple matches search

As said earlier, CDAWG is almost 3 times faster than Spine for a single prefix search and But the ratio is not same in case of multiple searches. Table 6.3 shows multiple search times for various data and query sequence. Here we can verify the fact that CDAWG is about 1.5 times faster only. The reason lies in the fact that Spine processes a smaller number of suffixes which is possible due to smart link management strategy.

Memory usage

Lets now move on to see how much cost we are paying in terms of memory resources while using these index structures. Figure 6.4 shows the bytes consumed for indexing one character of indexed string. Here we can see that Spine clearly beats CDAWG and shows its utility in the fixed budget scenario. While the consumption in Spine for each single character is about 12 bytes, CDAWG consumes as much as 21 bytes per character. This clearly indicates that Spine can index approximately *40% longer* strings as compared to CDAWG given a particular amount of memory.

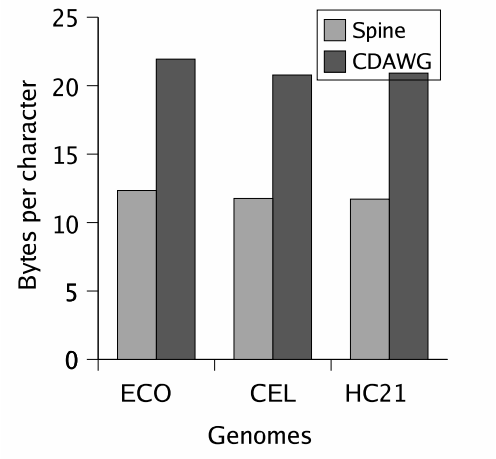


Figure 6.4: Memory Usage (Bytes per character)

6.3 On-Disk performance

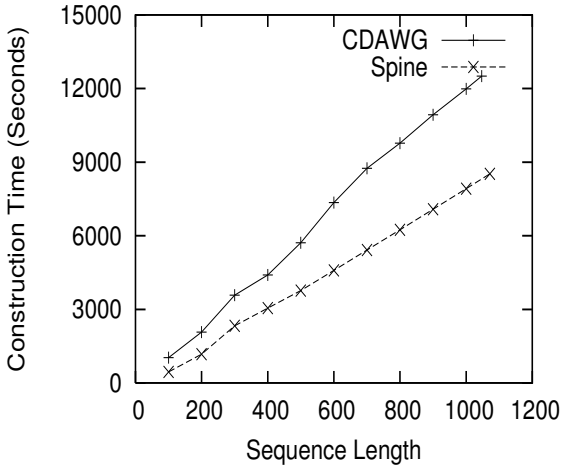


Figure 6.5: Index Construction Time on Disk

We will now move to on-disk performance analysis of Spine and CDAWG. The disk based implementation was done without any disk optimization. It means each of the write to the index structure goes directly to the disk[2].

Construction times of both the structures for varying size of indexed sequences are shown in Figure 6.5. We can clearly see that Spine is almost *1.5 times* faster than CDAWG. The advantage of smaller

[2]This is done through opening the index file with O_SYNC option which makes each write to the file result in a disk write.

node size of Spine become significant in this scenario as it reduces the disk seek time. On further investigation, we found that the number of write operations performed by Spine is much less than that of CDAWG. This shows that the number of changes made in Spine structure on addition of a new character is smaller than that of CDAWG. As disk write operation is more costly than any other operation (e.g. comparison), this offsets the advantage of CDAWG of having less number of comparison operations. The numbers of write operations for both structures are shown in Figure 6.6.

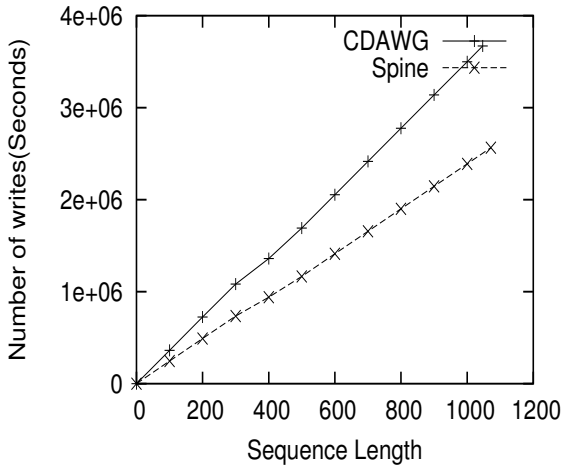


Figure 6.6: Number of Disk Write operation during index construction

Searching times[3] for both the disk indexes are shown in Table 6.4. Experiments are done exactly in the same manner as in case of in-memory environment. We can see that search times for both the indexes are almost equal. Here, we can see the performance of Spine is better than that of in-memory environment. This improvement is expected because of smaller node size in case of Spine. But the improvement is not as much as in case of construction time on disk. The reason for poor performance of CDAWG was higher numbers of disk writes which are absent in case of search operations. Here, CDAWG takes the advantage of fewer comparisons (and hence fewer disk reads) which offsets the longer seek time.

6.4 Node Size Optimizations for CDAWG

Now we suggest a couple of optimizations on the basis of observations during experimentation. The optimizations are similar as suggested in **??** for reducing node size.

[3]Only first 1M characters of HC21 are considered here

Indexed Seq.	Query	Spine	CDAWG	Avg mLen
ECO	CEL	21.41	19.71	10.80
ECO	HC21	21.13	18.62	10.57
HC21	ECO	19.95	17.42	10.03
CEL	ECO	25.14	22.17	11.43

Table 6.4: Average search times(Microseconds) for single prefix search on Disk

6.4.1 Small Edge Length

This is one of the simplest observation that all the edges other than open edges have length less than or equal to 25000 with all the datasets, and hence it can be represented by a short integer. For open edges, we need not to store target node for an open edge, hence the target field can be used as length field for open edges. Maximum edge lengths for various datasets are shown in Table 6.5. This simple optimization results in saving of 8 bytes per state and hence approximately 4 bytes per character.

6.4.2 Sparse edge distribution

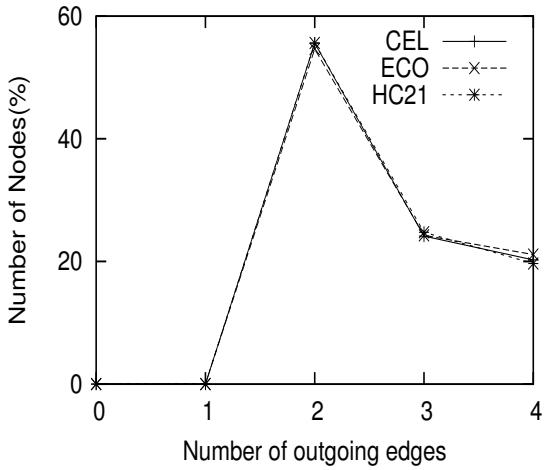


Figure 6.7: Edge Distribution

Sequence	Max Edge Len
ECO	1772
CEL	8126
HC21	21831

Table 6.5: Maximum Edge Length

The Figure 6.7 shows graph of the number of nodes with the number of outgoing edges. We can see that almost 50% of nodes are having just two edges and only about 25% of nodes have all the four edges. This observation suggests a multi-table node structure similar to Spine, though simpler in this case.

Chapter 7

EXTENSION FOR ENGLISH TEXT

After doing experiments with genomic sequences (with $|\Sigma| = 4$), we also extended both the structures for the English text (with $|\Sigma| = 26$) to analyze the performance of these index structure with larger alphabets as well. These structures are extended as follows.

7.1 Spine

In the Spine implementation for genomic strings, the entire structure has been partitioned into five tables. This was done to utilize very sparse rib distribution for reducing space requirements of the structure(see [12] for details). The final node lay out in such case is shown in Figure 7.1. In figure LT denotes Link Table, RT denotes rib table, LD is Link Destination, RD is rib destination, PT is Parent threshold value and finally PRT is the PRT value for the extrib.

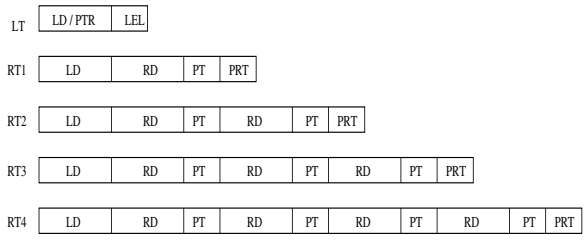


Figure 7.1: Final Node Layout for Genomic Sequences

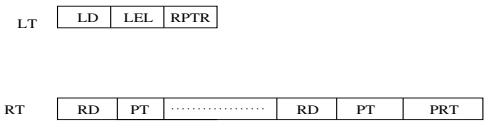


Figure 7.2: Final Node Layout for Text Sequences

When the alphabet size is not fixed , this node layout is not acceptable because of its static nature.

Alternatively we can implement it using single table only and dynamically allocating ribentry whenever required. Figure 7.3 shows the distribution of ribs for three text datasets with $|\Sigma| = 26$. From the figure it is clear that only 20 to 30% of nodes have ribs associated with them, i.e. size overhead is not very large. Hence final node lay out comprises of one rib table with each of the entry as a ribentry and one link table with each of the entry as node entry. These structures are shown in Figure 7.2. RPTR is a pointer to RibEntry which is stored in RT. Other symbols are same as before.

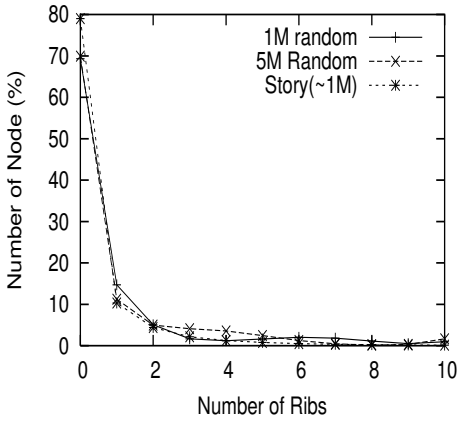


Figure 7.3: Rib Distribution

7.2 CDAWG

As discussed earlier, CDAWG for genomic strings has been implemented using transition matrix. The transition matrix based implementation is good for genomic strings as alphabet size is smaller in such cases, and as a result size of node is same as that of adjacency list based implementation. But, in case of larger alphabet result will be too much wastage of memory. Memory requirement can be made minimal using adjacency list based implementation, but the search queries will become slower.

Before deciding the final layout, lets see the distribution of outgoing edges from a state (Figure 7.4). We can see that around 50% of the nodes have two outgoing edges. We can use this observation to reduce the size of the node. The first two edges of a node are stored with the node itself. Each edge can be identified by its label. So we lose the advantage of transition matrix. But if a node has more than two edges then the edges are stored in a separate table and node has a pointer to it. Each entry of this table will consist of $|\Sigma|$ edges, hence we can directly index to a particular edge instead of searching it through it's label. As clear from graph, this will require only 50% of nodes to have a entry in table.

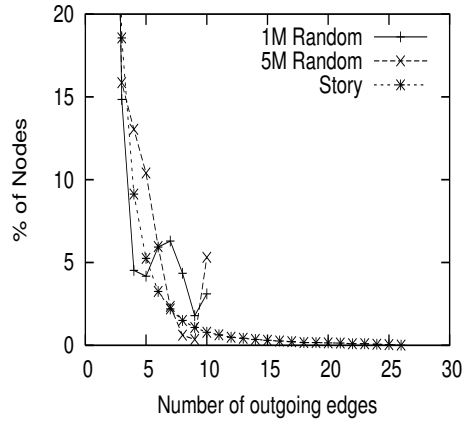


Figure 7.4: Edge Distribution in CDAWG

Indexed Seq.	Spine	CDAWG
1M Random	1.1	1.3
5M Random	6.3	7.7
Story	0.9	0.7

Table 7.1: Average Construction times(Seconds)

7.3 Performance

For analyzing the performance of the two indexes for text, we generate two random sequences of size 1M and 5M characters resp. Other than these two sequences, we use one real English story of size approximately 1M characters. The construction times and search times for these sequences are shown in Table 7.1 and Table 7.2. The trends are very similar to what were obtained in case of in-memory environment for genomic sequences.

Indexed Seq.	Query	Spine	CDAWG	Avg mLen
Story	1M Rnd	4.1	0.6	2.67
5M Rnd	Story	3.78	0.34	0.76
5M Rnd	1M Rnd	4.20	2.22	6.45

Table 7.2: Average search times(Microseconds) for single prefix search

Chapter 8

INTEGRATION WITH BLAST

BLAST(Basic Local Alignment Search Tool) [15] is a popular tool for doing the local alignment of nucleotides and proteins. The functionality of BLAST can be divided in two parts, finding exact matches and then extending these matches to maximize the scoring function. The BLAST algorithm can be summarized as follows[1]:
The BLAST algorithm is a heuristic search method that seeks words of length W that score at least T when aligned with the query and scored with a substitution matrix. Words in the database that score T or greater are extended in both directions in an attempt to find a locally optimal ungapped alignment or HSP (high scoring pair) with a score of at least S or an E value lower than the specified threshold. HSPs that meet these criteria will be reported by BLAST, provided they do not exceed the cutoff value specified for number of descriptions and/or alignments to report.

In an abstract view, BLAST algorithm can be seen as shown in Figure 8.1. During first phase, BLAST does not use any index to find out the initial matches and takes an heuristic approach instead, which affects the quality of answer. If the value of W is higher (e.g. 11 for nucleotides) the performance of BLAST is satisfactory in terms of time, but the quality of result is not. It may miss few interesting matches. The loss becomes significant if the aligned sequences are highly different or if smaller matches are also of biological significance. [14] shows retrieval effectiveness of BLAST with W=11 as 93%, which can be improved by reducing the value of W. But when the value of W is on the smaller side (e.g. less 10 for nucleotides), the time taken by BLAST increases very rapidly. a string index structure can be integrated with the BLAST to improve the running time as performance of an index does not get affected by the match length significantly. We integrated both the index structures with BLAST. Details are discussed in the following section.

[1] As given at [16]

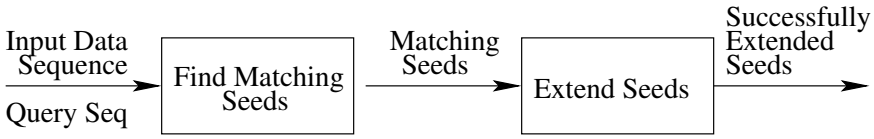


Figure 8.1: Pictorial representation of BLAST algorithm

8.1 Implementation details

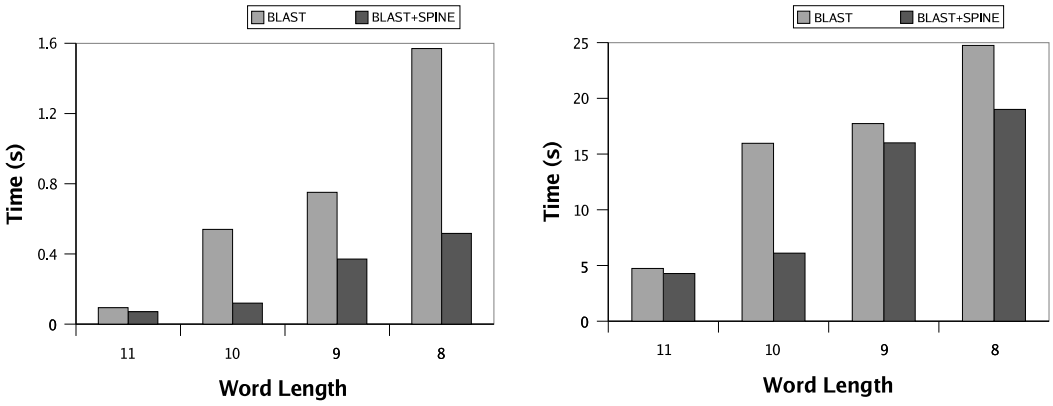
The BLAST code has been taken from the NCBI repository. The code which deals with the initial matching process contains around 11000 lines of code (namely, blast.c file). We integrated Spine and CDAWG through a common interface. The index is called from the ncbimain() function and the all match positions of length greater than W are stored. Now, we have to use these seeds in the next phase of BLAST. BLAST code does not have a clearly divided two phases so extreme care has to be taken so that we should disturb the execution of second phase. BLAST uses the function BlastNtWordFinder() given in blast.c, just before second phase of algorithm to find the initial matches and provide them as input to second phase. Now, at this point, we have to use the previously stored seeds and convert them to the format which is compatible to second phase. As said earlier that the boundary between phase one and second is not clear e.g. BlastNtWordFinder() is invoked for each of the 5M chunks, but we provide seeds for the entire sequence at once, hence it should be appropriately handled.

8.1.1 Experimental Analysis

Total running times of BLAST with and without index are shown in Figure 8.2 for various initial seed length(W). We are showing result for two datasets. First one as Vibrio cholerae chromosome as input data sequence, which is approximately 1MBP[2] in size. The query taken was a 5KBP prefix of C.Elegans earthworm genome. Second one has E.Coli bacterial genome of 3.5 MBP as data sequence and Vibrio cholerae chromosome as query.

For the default value of W (i.e. 11) there is no significant improvement but as the seed length W decreases, performance improvements become more and more significant. This trend can be explained as follows. When the value of W is small, due to heuristic search method, BLAST reports a longer match multiple time as multiple small matches of length W starting at different places in the same longer match.

[2]MBP : Million BasePairs



(a) (b)

Figure 8.2: Running times of BLAST and BLAST+SPINE Index(Seconds) for **(a)** Vibrio cholerae chromosome as data sequence and 5K prefix of C.Elegans as Query **(b)** E.Coli as data sequence and Vibrio cholerae chromosome as Query

This results in both longer search time and multiple extensions. All the extensions returns same result and then are of no use. But in case of index being present, only the longest possible matches are reported which reduces the number of extensions in second phase of execution. Again, search time for index is not sensitive to parameter W and hence, it reduces execution time of first phase as well.

Chapter 9

CONCLUSION

In the presented work, we mainly focused on comparison of Spine index with CDAWG, extension and evaluation of spine for larger alphabet and integration of Spine and CDAWG with BLAST. After doing a course of experiments with Spine and CDAWG, we conclude:

1. In in-memory environments, CDAWG was observed to be approximately 3 times faster than Spine for longest matching prefix search queries. Also, for all pair multiple search queries CDAWG is faster but by a factor of 1.5 this time.

2. Spine is much more memory friendly as compared to CDAWG and shows that it can index approximately 40% longer strings than that of CDAWG.

3. On disk, Spine takes advantage over CDAWG because of its smaller node size, and hence provides a 1.5 times faster index construction. But for search queries on disk, both the structures perform almost equally.

4. Both structures show high locality of references as most of the link's (suffix link's in CDAWG) targets are clustered in the top part of the index.

5. Performance of both the structures for text alphabet ($|\Sigma| = 26$) also remains same which shows the scalability in terms of alphabet size also.

6. The integration of indexes with BLAST shows that run time performance of BLAST can be improved for smaller seed value as well to achieve high quality of results.

In short, we can say if enough resources are available then its better to use CDAWG but if there are constraints on resources, Spine is a better option.

Bibliography

[1] S. Bedathur and J. Haritsa. *Engineering a Fast Online Persistent Suffix Tree Construction.* Proc. of 20th IEEE Intl. Conf. on Data Engineering (ICDE), Boston, USA, March 2004, pgs. 720-731

[2] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen and J. Seiferas. *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40:31-55, 1985

[3] A. Blumer, J. Blumer, D. Haussler, R. McConnell and A. Ehrenfeucht. *Complete inverted files for efficient text retrieval and analysis.* Journal of the Association for Computing Machinery, 34(3):578-595, July 1987

[4] M. Crochemore and R. Verin. *On compact directed acyclic word graphs.* In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, Structures in Logic and Computer Science, volume 1261 of Lecture Notes in Computer Science, pages 192–211. Springer-Verlag, 1997.

[5] R. Giegerich, S. Kurtz and J. Stoye. *"Efficient Implementation of Lazy Suffix Trees.* Proc. of 3rd Workshop On Algorithm Engineering, July 1999.

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

[7] E. Hunt, M. Atkinson and R. Irving. *A Database Index to Large Biological Sequences.* Proc. of the 27th VLDB Conference, 2001.

[8] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi.*On-line construction of compact directed acyclic word graphs.* In Proc. 12th Annual Symposium on Combinatorial Pattern Matching, July 2001.

[9] S. Kurtz.*Reducing the space requirements of suffix trees.* Software Practice and Experience, 29:1149-1171, 1999.

[10] U. Manber and G. Myers. *Suffix Arrays: A New Method for On-line String Searches*. SIAM J. Comput., 22(5), 1993.

[11] E. McCreight. *A space-economical suffix tree construction algorithm*. Journal of the ACM, 23(2):262-272, Apr. 1976.

[12] N. Neelapala, R. Mittal and J. Haritsa.*SPINE: Putting Backbone into String Indexing*. Proc. of 20th IEEE Intl. Conf. on Data Engineering (ICDE), Boston, USA, March 2004, pgs. 325-336

[13] E. Ukkonen. *On-line construction of suffix trees*. Algorithmica, 14:249-60, 1995.

[14] H. Williams, J. Zobel. *Indexing and Retrieval for Genomic Databases*. In IEEE Transactions on Knowledge and Data Engineering, 14(1):63-78, 2002.

[15] http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/information3.html

[16] http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/BLAST_algorithm.html

[17] http://www.nist.gov/dads/HTML/trie.html

Appendix A

Online Linear Construction Algorithm for CDAWG

This algorithm has been taken from [8].

Algorithm

function Build_CDAWG(x$)
Given alphabet $\Sigma = x[-1], x[-2], .., x[-m]$
/* $ is the end-marker appearing nowhere in x. */
1 create nodes $source$, $sink$, and $\perp$;
2 **for** $j := 1 \, to \, m$ **do**
. create a new edge $(\perp; (-j; -j); source)$;
3 $Suf(source) := \perp$;
4 $length(source) := 0$; $length(\perp) := -1$;
5 $e := 0$; $length(sink) := e$;
6 $(s, k) := (source, 1)$; $i := 0$;
7 **repeat**
8 $i := i + 1$; $e := i$; /* e is a global variable. */
9 $(s, k) := update(s, (k, i))$;
10 **until** $w[i] = \$$;

function update(s; (k; p)): pair of node and integers;
/* (s; (k; p - 1)) is the canonical reference pair for the active point. */

```
1   c := w[p]; oldr := nil;
2   while(not check_end_point(s; (k; p − 1); c)do
3         if k ≤ p − 1 then /* implicit case. */
4               if s' = extension(s; (k; p1))then
5                     redirect_edge(s; (k; p − 1); r);
6                     (s; k) := canonize(Suf(s); (k; p − 1));
7                     continue;
8               else
9                     s' := extension(s; (k; p − 1));
10                    r := split_edge(s; (k; p − 1));
11        else /* explicit case. */
12              r := s;
13        create edge (r; (p; e); sink);
14        if oldr ≠ nil then suf(oldr) := r;
15        oldr := r;
16        (s; k) := canonize(suf(s); (k; p − 1));
17  if oldr ≠ nilthen suf(oldr) := s;
18  return separate_node(s; (k; p));
```

function extension(s; (k; p)): node;

/* (s; (k; p)) is a canonical reference pair. */

```
1   if k > p then return s; /* explicit case. */
2   find the w[k] − edge (s; (k'; p'); s') from s;
3   return s';
```

function redirect_edge(s; (k; p); r);

```
1   let (s; (k; p'); s') be the w[k]-edge from s;
2   replace the edge by edge (s; (k'; k' + p − k); r);
```

function split_edge(s; (k; p)): node;

```
1   let (s; (k'; p'); s') be the w[k]-edge from s;
2   create node r;
3   replace the edge by (s; (k'; k' + p − k); r) and (r; (k' + p − k + 1; p'); s');
4   length(r) := length(s) + (p − k + 1);
5   return r;
```

function separate_node(s; (k; p)): pair of node and integer;

1 $(s'; k') := canonize(s; (k; p))$;
2 **if** $k' \leq p$ **then return** $(s'; k')$; /* implicit case. */
3 /* explicit case. */
4 **if** $length(s') = length(s) + (p - k + 1)$ **then return** $(s'; k')$;
5 create node r' as a duplication of s' with the out-going edges;
6 $Suf(r') := Suf(s')$; $Suf(s') := r'$;
7 $length(r') := length(s) + (p - k + 1)$;
8 **repeat**
9 replace the $w[k]$-edge from s to s' by edge $(s; (k; p); r')$;
10 $(s; k) := canonize(Suf(s); (k; p - 1))$;
11 **until** $(s'; k') \neq canonize(s; (k; p))$;
12 **return** $(r'; p + 1)$;