The SPINE Genomic Index

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

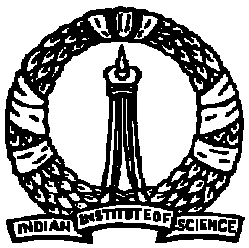
Master of Engineering

in

Faculty of Engineering

by

Naresh Neelapala and Romil Mittal



Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

MARCH 2003

Contents

List of Figures

Chapter 1

Abstract

We present SPINE, an index structure for genomic databases. SPINE features horizontal compaction of tries resulting in a backbone formed by a linear chain of nodes representing the underlying genome sequence. The nodes are connected by a rich set of edges for facilitating fast forward and backward traversals over the backbone during index construction and query search.

In this paper, we describe algorithms for SPINE index construction and for searching this index to find the occurrences of query patterns. Our experimental results on a variety of real and synthetic genomic sequences show that apart from being space-efficient, SPINE takes lesser time for both construction and search as compared to the classical suffix tree based indexes, both in-memory and on-disk. We also demonstrate how SPINE's search performance is orders of magnitude faster than BLASTN, the popular sequence alignment tool.

Chapter 2

Introduction

Biological sequence data, including DNA and protein sequences, is being produced at a rapid rate and quantity by geneticists and medical researchers. These sequences are stored in large libraries such as NCBI Genbank [14], EMBL [15], Swissprot [17], etc. An extremely common operation on these libraries is to search over them looking for various kinds of biologically useful information, such as motifs, local and global alignments, physical and genetic maps, etc. For example, a biologist might wish to compare the rat genes involved in hypertension against the human and mouse genomes in order to establish the correspondence between their genetic sequences.

Effective searching is hampered by the size and number of genome sequences – for example, it is estimated that the volume of data at NCBI Genbank, the world's largest repository of sequences, *doubles* every year. This is not surprising when we consider that mammalian genomes such as the human genome are typically around 3 billion base-pairs in length [10]. Further, the number of queries on these databases has also seen an explosive growth. Therefore, there is a clear need for developing mechanisms that can significantly speed up search-based operations on large sequence databases.

State-of-the-Art

The sequence search tools that are currently used by biologists can be broadly classified under two heads [13]: *Seed-based*, exemplified by BLAST, the classical sequence alignment package [7], and *SuffixTree-based*, exemplified by MUMmer [6], the recently-developed alignment software from Celera Genomics and TIGR (The Institute for Genomics Research) .

In the seed-based approach, the data sequence is first searched for exact-matches of short *seed sequences* [1] from the query sequence. These seed sequences are stored in a keyword tree that is usually implemented as a hash table. The successful exact matches then form the candidates that are extended into better alignments [7].

In the SuffixTree-based approach, on the other hand, an explicit index called the *suffix tree* [1] is created for the entire data sequence – this index stores all suffixes of the data sequence in a vertically-compacted trie structure. The popularity of suffix trees can be ascribed to their having linear (in the size of the data) construction time and space complexity as well as linear (in the size of the query) searching times.

While almost all the current tools fall under one of the above approaches, recently a two-level search technique called MAP was proposed in [12], wherein a preprocessing phase using an approximate index is used to first filter out those regions of the data sequence that potentially contain matching entries, and then a seed-based approach is used on the filtered regions.

The SPINE Index

We present in this paper a new index structure, called **SPINE** (Sequence Processing INdexing Engine), which has a variety of advantages with regard to the previous approaches in terms of its search performance. The quantitative improvements are demonstrated

[1] It has been empirically determined that a seed length of 11 yields good results for DNA sequences.

by comparing SPINE against both BLAST and MUMmer for a variety of real and synthetic genomic sequences. Specifically, our experimental results indicate that for memory-resident indexes, SPINE is about thirty percent faster with regard to MUMmer, while for disk-based implementations, the gap is substantially more. With respect to BLAST, we find that SPINE provides search times that are about three to four orders of magnitude faster (in [11], MAP was reported to be between one to two orders of magnitude faster than BLAST).

A sample picture of a SPINE index is shown in Figure 2.1 for the data sequence `aaccacaaca`. At its core, the SPINE index consists of a *backbone* formed by a linear chain of nodes connected by *vertebra* edges, representing the underlying genome sequence. The nodes are additionally connected by forward *ribs* and *extension ribs*, and backward *links* for facilitating fast traversals over the backbone during the index construction and query search processes. All the edges have associated labels that are assigned during the construction process and are used to determine which paths are valid for traversal in the SPINE structure.

From an abstract viewpoint, SPINE can be viewed as a **horizontal compaction** of the trie of the data sequence, in marked contrast to suffix trees which represent, as mentioned earlier, a *vertical* trie compaction. The motivation behind this horizontal compaction is to avoid the duplication of common segments among the various paths in the trie, thus reducing the number of nodes and thereby the space required to index a sequence. In fact, it carries this to the logical extreme of representing each character of the original data sequence only *once* in the index structure. This is in contrast to the suffix trees, where the number of nodes may go upto *double* the number of characters in the sequence.

Further, the improvement is not restricted to just the number of nodes, but the *size* of SPINE nodes is also smaller than their suffix-tree counterparts. The average index overhead per sequence character never exceeded 12 bytes in all our experiments with SPINE, whereas MUMmer requires 17.4 bytes per character indexed. We hasten to add here that there is a rich body of literature that has attempted to reduce the index overhead

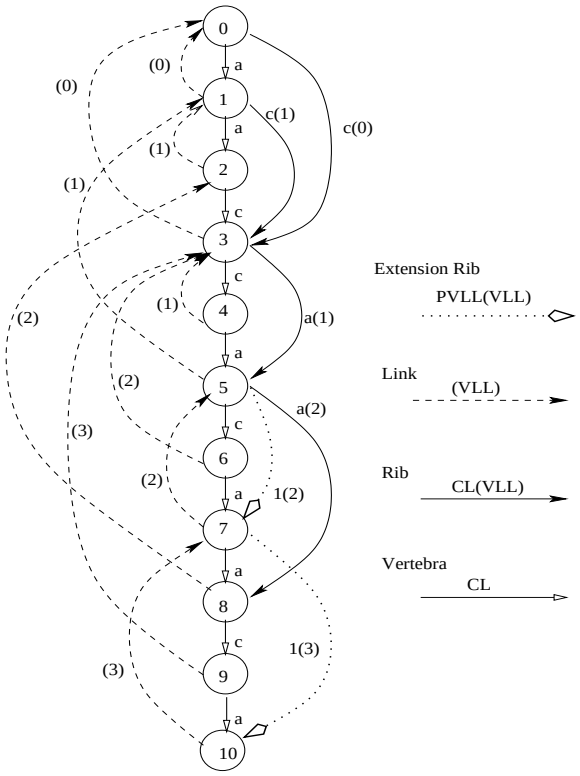


Figure 2.1: **Example SPINE Index (for *aaccacaaca*)**

of suffix trees – for example, Lazy Suffix Trees [4] and Kurtz Suffix Trees [3] require only 8.5 bytes and 12.5 bytes per indexed character – however, as described in detail later in Chapter 3, such optimizations typically adversely impact either the performance or the supported functionality, and are therefore not viable from a general usage perspective.

The horizontal compaction strategy results in SPINE being able to handle larger sequences than suffix trees for a given memory budget. And when it comes to disk-resident indexes, SPINE exhibits significantly higher node locality and much fewer edge traversals, making it attractive for disk-based implementations. Further, due to the simplicity of its structure, it appears easier to develop good buffering policies for SPINE, as compared to suffix tree based indexes.

Chapter 3

Related Work

A rich body of literature exists with regard to optimizing the space occupied by suffix-trees – however, these optimizations typically adversely impact either the performance or the functionality. For example, Kurtz [3] proposed an implementation that requires 12.5 bytes per character indexed for DNA sequences. However, although the space required is less, suffix trees built using this technique take more time for construction and searching times are not good as well due to comparatively larger number of edge traversals. An even more space-efficient implementation, called Lazy Suffix Trees [4], was recently proposed, taking only 8.5 bytes per character indexed. However, it has constraints on its functionality, including not being online, and not being able to perform approximate and subsequence matching efficiently due to the absence of suffix links. Finally, suffix arrays[9] reduce the space requirement to just 6 bytes per indexed character but increase the time complexity from linear to $O(d\ log\ d)$, where d is the sequence length.

A related class of indexes is *DAWGS - Direct Acyclic Graphs* [8], but these indexes lack position information of the matching pattern in the data sequence. Also, it is not possible to do approximate matching and finding multiple occurences of a pattern efficiently.

In order to make suffix-tree construction on disk efficient, a partition-based technique was recently proposed in [10]. For this algorithm to work, it requires dispensing completely with the suffix links that are essential for retaining the linear time complexity – as a result, the algorithm in [10] has quadratic complexity. Further, the removal of the

links makes approximate matching and subsequence matching rather inefficient. Finally, this is not an online algorithm – that is, if a new character is added to the sequence, the entire index has to be rebuilt.

Chapter 4

Contributions

In this report, we describe the complete SPINE index structure. We also present algorithms for SPINE index construction and for searching this index to find the first occurrence as well as all occurrences of a query pattern. Further, the methodology to find all matching subsequences between two given sequences is described. The performance of SPINE is compared against MUMmer and BLASTN over a variety of real and synthetic genomic sequences, for both memory-resident and disk-resident scenarios.

Chapter 5

Organization

The remainder of this paper is organized as follows: In Chapter 6, the SPINE structure is presented in detail. The SPINE construction algorithm is described in Chapter 7 with a correctness proof in Chapter 8, and the searching algorithms are described in Chapter 9. The specifics of our prototype implementation are outlined in Chapter 10. Experimental results on the performance of this prototype are highlighted in Chapter 11 and 12 for memory-resident and disk-resident indexes, respectively. Finally, in Chapter 13, we summarize the conclusions of our study and outline future avenues to explore.

Chapter 6

The SPINE Index Structure

The central component of SPINE is the "backbone"(Figure 2.1). The backbone comprises of a chain of vertebras. Each vertebra on the backbone corresponds to a character in the input data sequence, 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 sequence, and their edge direction is always *downstream*.

While the backbone forms one source of connectivity between the nodes, there are additional directed edges that connect nodes across the backbone. There are two varieties of forward or downstream edges, called "ribs" (full lines in Figure 2.1) and "extension ribs" (dotted lines in Figure 2.1), respectively. These forward edges represent (in conjunction with the backbone) all possible suffixes of the data sequence, and are used during the query search process. Each rib is also labeled with a character label, corresponding to the character that it represents in the associated suffix.

The backward or upstream edges, called "links" (dashed lines in Figure 2.1) are created and used during the construction process of SPINE. Also, they aid in performing some of the search operations more efficiently.

The SPINE index in Figure 2.1 represents the horizontal compaction of all the suffixes in the corresponding Trie shown in Figure 6.1. In this compaction that SPINE brings to bear, all matching paths in the trie are merged into a single path. For instance, *ccacaaca* appears thrice in the trie but only once in SPINE. This means that all paths that were

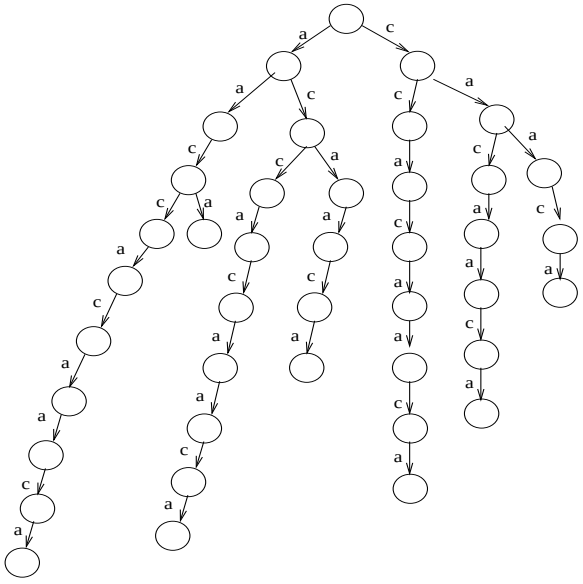


Figure 6.1: **TRIE (for *aaccacaaca*)**

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 sequences, may arise. For instance, in Figure 2.1, a path for aacaac appears to exist in the SPINE index even though it is not a subsequence of the given data sequence.

To avoid such false positives, we take recourse to a novel labeling strategy for the ribs, extension ribs, and links. Specifically, each rib, extension rib and link has a numeric label (*Link Label(LL)* with the links and *Validity Label(VL)* with the ribs and extension ribs) attached to it. These labels are assigned during the construction process and are used while traversing the index during the search process to avoid false positives. For example, in Figure 2.1, the rib leading out of the topmost node has the label c(0), with the number in brackets (in this case, 0) representing the validity length. Similarly, the link from the node 3 to the top node has a validity length of 0.

In the remainder of this section, we describe the components of SPINE in detail and the associated terminology. Our discussion assumes that the data sequence is composed of n characters.

6.1 Vertebra Backbone

The backbone is initially created with a single node, called the *root node*, and for each character in the data sequence, a new node is added at the bottom of the backbone using a vertebra edge labeled with the corresponding character. The node that is currently at the bottom of the chain is referred to as the *tail node*, N_{tail}. Each node has an integer identifier, with the identifier indicating the position of the node with respect to the sequence. That is, a node's identifier is equal to the length of the sequence above the node. This identifier is not stored in the node, but is used merely for representation. With this naming convention, the root node has identifier 0, and the tail node of the entire sequence has identifier n. From now onwards, we will refer to a *node* i as N_i.

We now define some terminology that we will use to describe the remaining components of the data structure: S_i denotes the string formed by the concatenation of the characters on the backbone from the root node to N_i. $AllSuf_i$ denotes the set of all suffixes of S_i, that is, $AllSuf_i = \{s_{i1}, s_{i2}, s_{i3}, s_{i4}, \ldots, s_{ii}\}$ where s_{ij} is a suffix of S_i of length j, and so s_{i1} is a suffix of s_{i2}, and so on. With this definition, the set of all suffixes for the complete sequence (of length n) is denoted by $AllSuf_n = \{s_{n1}, s_{n2}, s_{n3}, s_{n4}, \ldots, s_{nn}\}$. Finally, $EndSuf_i$ denotes the set of suffixes of S_i ending at N_i (i.e. paths exist for them from the root node to N_i), while $EndSuf_i(k)$ denotes the set of suffixes in $EndSuf_i$ that are of length k or less.

6.2 Links

Consider the following situation: Let s be a common suffix of S_n and S_m, where S_m is a prefix of S_n, that is, $s \in AllSuf_m \ \wedge \ s \in AllSuf_n \ (m < n)$. Then s will not terminate at N_n, which means that $s \notin EndSuf_n$. Now assume that s_{nk} is the longest suffix of S_n which does not end at N_n. In other words, s_{nk} is the longest suffix which had been previously created and now also features in the suffixes of S_n. This implies that $\exists p | (p < n) \ \wedge \ (s_{nk} \in EndSuf_p)$, i.e. there exists some upstream node N_p where s_{nk} terminates.

Whenever the above situation occurs, a link is constructed from N_n to N_p with a $LL = k$.

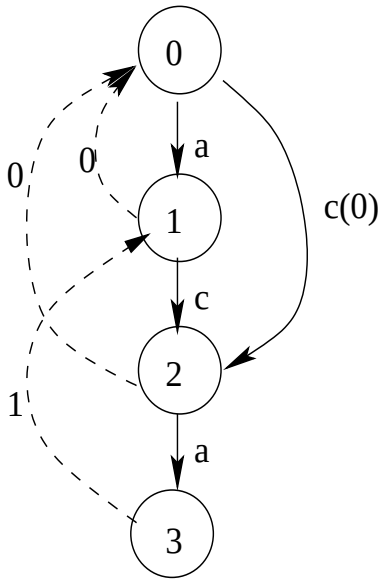


Figure 6.2: **Links (for sequence aca)**

An example of link construction is shown in Figure 6.2 for the sequence aca. Here, starting from the root, all the suffixes of aca, except a, end at N_3, i.e.

$$AllSuf_3 = \{\underbrace{a}, \underbrace{ca, aca}\}$$
$$EndSuf_3 = \{ca, aca\} \text{ and } a \in EndSuf_1$$

Therefore, the link at N_3 points to the node which indexes a, i.e. N_1. The details of how the labels are assigned to links are discussed later in Chapter 7.

Figure 4 gives an algorithm **ALLSUFGEN** to show how $AllSuf_i$, the set of all suffixes of the sequence ending at N_i, can be recovered by traversing the link chain starting at N_{i-1}. In this algorithm, $LinkLabel(j)$ is a function that returns the LL associated with the link emanating from N_j, while $LinkDest(j)$ is a function that returns the identifier of the node pointed to by the link emanating from N_j. $LinkDest(j)$ returns $NULL$ for a node from which there is no link, i.e. the root node. This means that the algorithm terminates when the root node is reached.

ALLSUFGEN(i)

1. $AllSuf_i = EndSuf_i$
2. $l = LinkLabel(i-1)$;
3. $j = LinkDest(i-1)$;
4. while $(j \neq NULL)$
5. {
7. $AllSuf_i = AllSuf_i \cup EndSuf_j(l)$;
8. $l = LinkLabel(j)$;
9. $j = LinkDest(j)$;
10. }
11. return $AllSuf_i$

Figure 6.3: **ALLSUFGEN**

6.2.1 Important observation regarding LLs

As described, the link label l for a link from node a to node b represents the maximum length of the suffix s which is indexed by node b instead of node a. But s would again be a suffix of the all suffixes ending at node a. This implies that s would exist above both nodes a and b. So, this can be stated as, "A sequence of length l would be same above node a and node b, if there exists a link with LL equal to l from node a to node b". For instance, in Figure 2.1, there is a link with LL=3 from node 9 to node 3. And we can observe that a pattern of length 3 *aac* is present above both node 3 and node 9.

Looking at the LL values and the maximum LL value for the sequence, one can infer the degree of repitition of patterns in the sequence. And this could be a very useful information for the biologists who just want to study particular sequences.

The above mentioned observation would be used to find the multiple occurences of a pattern in a given sequence. The methodology will be described in Chapter 9.

6.3 Ribs

Suppose SPINE is built for S_j i.e. the first j characters of the sequence. Now to construct SPINE for S_{j+1} from S_j (to append the $(j+1)^{th}$ character), we need to extend all the suffixes of S_j by this $(j+1)^{th}$ character. The newly added character, c_{tail}, on the backbone automatically extends the suffixes that feature in $EndSuf_j$. For the remaining suffixes in $AllSuf_j$, however, the appropriate upstream nodes are reached by traversing the link chain. If a rib/vertebra does not already exist for c_{tail} at any node in the link chain, a new **rib** is created from that node to the newly-created node, N_{tail}.

The traversal of the link chain terminates if any one of the following conditions occur :

- the root node is reached
- a node having an outgoing vertebra labeled with c_{tail} is reached
- a node having an outgoing rib labeled with c_{tail} is reached

The first stopping condition is obvious since no further traversal is possible, while the other two conditions reflect the fact that the suffix in question has *already* been extended.

When a new rib is created originating from a node, its CL is set to c_{tail} and its VL is set to the length of the longest suffix of S_{tail-1} ending at that node, which is given by the LL of the last traversed suffix link. A rib is valid to traverse from all the suffixes with length $\leq$ VL of the rib. Intuitively, the rib VL represents the length of the longest prefix that can be traversed from the root before the rib is traversed. This means that while traversing SPINE, a rib can be traversed only if the length traversed from the root to the node from which the rib emanates is $\leq$ VL of the rib.

6.4 Extension Ribs

As discussed above, ribs are used to extend the suffixes ending at any node to obtain newly created suffixes due to addition of a new node at the tail. And the suffixes which

are extended are indicated by the VLs of the ribs. Now if while extending the suffixes, we find that there already exists a rib with the same CL but whose VL is *less than* the length of the longest suffix which needs to be extended, then an *extension rib* is used to further extend the rib. For ease of presentation, we will hereafter refer to the extension ribs as *extribs*.

Each extrib has an associated *VL*, which is the length of the longest suffix it is extending, as well as a *"parent rib label" (PRL)*, which is the VL of the rib that it is extending. This means that at any node there can be extribs with as many different PRLs as the number of ribs ending at that node. And for each PRL there can be many different extribs corresponding to different lengths of suffixes they extend. And so, for a node the number of extribs will not be fixed which will result in a non-uniform node size.

We solve the above problem by maintaining the extribs in a *chained* fashion. That is, the first extrib in the chain is located at the destination node of the rib which failed the validity 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 we need to create an extrib, instead of creating it from the destination of the rib corresponding to its PRL, we traverse to the node at the end of the extrib chain, and then create a new extrib from this node to the tail node.

Creation of ExtRib

Whenever the Rib Validity Test fails, the extrib chain is first checked to see whether there exists an extrib with PRL same as the VL of the rib which failed the test, and a VL that is $\geq$ length of longest suffix that needs to be extended. If such an extrib is found then a new extrib need not be created because the suffix in question has already been extended. Otherwise, a new extrib is created at the end of the chain with the appropriate VL and PRL.

To help make the above discussion clear, we present a sample construction of extribs in Figure 6.4. The notation for the labels of extribs used in the figure is: PRL(VL).

Appending N_8 (Figure 6.4(a)) When extending the suffixes of string above N_7 by 'a',

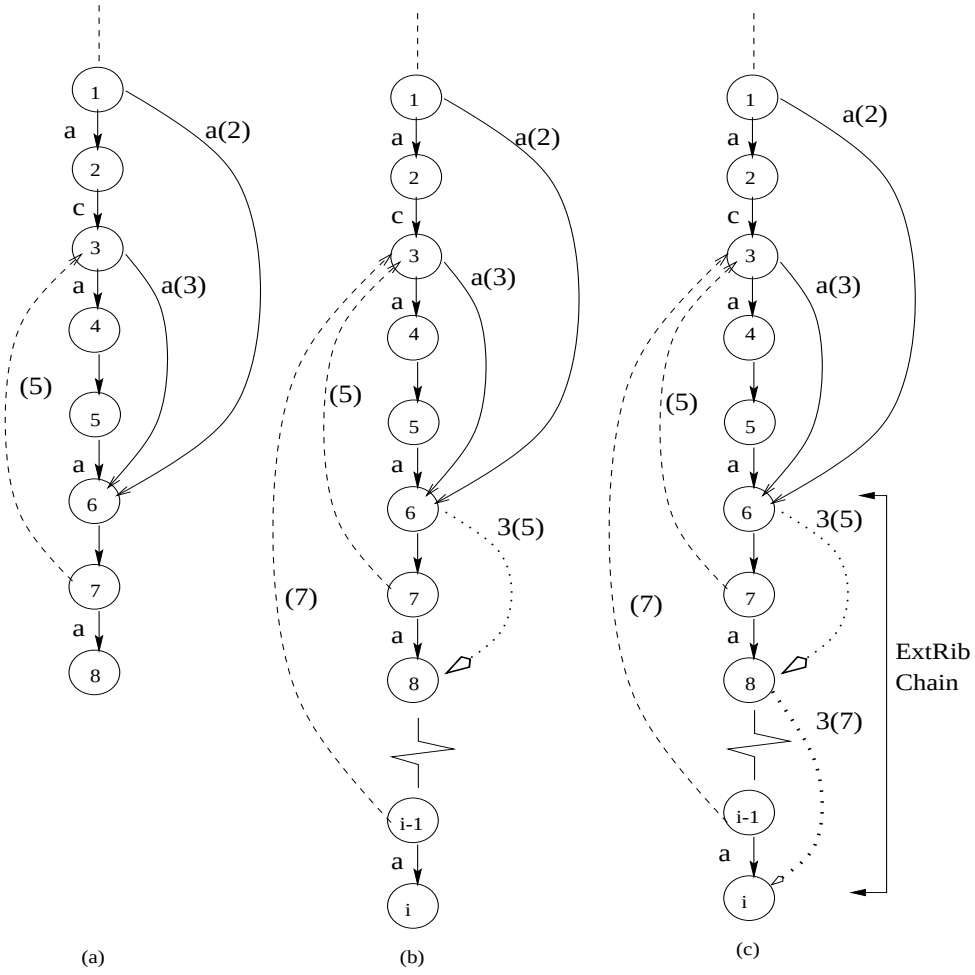


Figure 6.4: **Extension Ribs**

we need to extend all the suffixes retrieved by the ALLSUFGEN algorithm. In this process, after traversing the link at N_7, we see that the rib from N_3 to N_6, say R1, fails the validity test (VL of R1 = 3 < LinkLabel(N_7) = 5). Hence an extrib is created at N_6 with VL equal to 5 and PRL = 3 (VL of R1) (See Figure 6.4(b)).

Appending N_i (Figure 6.4(b)) In this case, we need to extend a suffix of length 7 (LinkLabel(N_{i-1})). So, R1 again fails the VL test (3 < 7). Hence the extrib chain from N_5 is traversed to find an extrib with $VL \geq 7$ and PRL = 3. Though the extrib from N_6 has the same PRL, it doesn't have the required VL. Hence, an extrib is created at the end of chain, i.e., at N_7 with its destination as N_i, VL as 7 and PRL as 3. Figure 6.4(c) shows the index that results after appending N_i.

Structural Invariants

Based on the above discussion on vertebras, ribs, extribs, links and their labels, we can state the following invariants of the SPINE data structure:

- Excepting the root, every node has exactly one link that points to an upstream node. Every node, except the tail node, has one vertebra pointing to the next node downstream. Given a character set with alphabet size K, the number of ribs emanating from a node can range between 0 and $K-1$. These ribs point to downstream nodes. In the case of DNA sequences, $K = 4$, and therefore the number of ribs is in the range $[0, 3]$. The tail node will always have zero ribs.

- A path from the root to a node represents the *first* occurrence of the sequence represented by this path, in the complete data sequence.

- A link from a N_j points to the N_i that indexes the longest suffix of S_j which is not indexed by N_j. The LL of the link represents the length of this longest suffix.

- All the suffixes till any node n are represented by the valid paths from the root to that node and to the nodes which can be reached using the link chain starting at node N_n.

- The ribs terminating at node i represent extensions to the suffixes of S_{i-1} to obtain the suffixes of S_i.

- The CL of a newly created rib is set to c_{tail} and its VL is set to the length of the longest suffix of the S_{tail-1}.

- The extribs are used to extend the suffixes ending at some node where there already exists a rib with the CL same as the new character being appended but a VL less than the required VL.

- There can exist only one extrib at any node, as any other extrib required at that node becomes a part of the extrib chain below that node.

Given this structure, we can show that the valid paths in the SPINE correspond exactly to the set of subsequences that occur in the data sequence – the detailed proof is given in Chapter 8.

Chapter 7

SPINE Construction Algorithm

In the last chapter, we presented an overview of the SPINE index structure. We now move on to presenting an *online* algorithm for constructing this structure. The entire pseudo code for the algorithm is given in Figure 7.1.

We start off with the SPINE initially consisting of just the root node and then, for each new character in the sequence, a node is appended to the tail of the SPINE. The vertebra connecting to the newly-added node is labeled with the new character, and the associated links and ribs are created as required.

7.1 Link/Rib/ExtRib Construction

As mentioned earlier, every node, excepting the root, has a link associated with it. When the first character is appended, a link with LL equal to 0 is created from the new node to the root node. For all subsequent nodes, the following process is followed: Link of the parent of N_{tail} is traversed upstream. Let the destination node of the current link be N_{curr}. At N_{curr} it is checked whether a vertebra/rib already exists for c_{tail}. If it is not present, a new rib is constructed from N_{curr} to N_{tail}. Then, the link at N_{curr} is traversed upwards and the same process is repeated with the new N_{curr}.

The above process stops with the creation of a new link, which happens when one of the following cases occur during the upward traversal of the link chain:

- **A vertebra is found with CL = $\mathbf{c_{tail}}$**. In this case, a link is created from N_{tail} to the destination node of the vertebra.

- **A rib is found with CL = $\mathbf{c_{tail}}$**. In this case, if the Rib Test does not fail, then a link is created from N_{tail} to the destination node of the rib. Otherwise, the extrib chain is traversed to find an extrib with VL $\geq$ LL of the link and belonging to the rib which failed the test. If found, then a link is created from N_{tail} to the destination of that extrib. Otherwise, a new extrib is created from the end of the extrib chain to the N_{tail} with appropriate PRL and a new link is also created from N_{tail} to the destination node of the last traversed extrib.

- **A rib is created from the root node**. Here, a link is created from N_{tail} to the root node.

Whenever a new rib/extrib/link is created, it is immediately assigned a VL (and also PRL for extribs). The method of assigning these labels is explained next.

7.2 Assigning Validity Labels

Ribs and Extension Ribs

A newly created rib or extrib is always given a VL equal to the LL of the last link traversed prior to its creation. An extrib is assigned a PRL equal to the VL of the rib which fails the validity test. To make the presentation uniform between ribs and extribs, we assume that a rib has an implicit PRL field which is always equal to its VL.

Links

Two cases exist while assigning a LL to a newly constructed link.

- When a rib or extrib is found with VL equal to or greater than the LL of the last link traversed, the new link is given a LL which is one greater than the LL of the last

link. An exception to this rule is that all links that point to the *root* are assigned a LL of 0.

- If a new link is constructed just after creating a new extrib, then it is assigned a LL which is one more than the VL of the last rib/extrib traversed whose PRL is same as the PRL of the newly created extrib.

7.3 Construction Example

To help clarify the above discussion, we now describe how the SPINE index is created for the same input sequence used in Figure 2.1, i.e. aaccacaaca. For better readability, the SPINE index is reproduced in Figure 7.2.

In the beginning, a root node is created with identifier 0. Subsequently, whenever a new node is added to the backbone, we start traversing the link chain beginning from the parent node of the newly added node. During this traversal, the various CASES highlighted in Figure 7.1 may arise, and we discuss each of them below:

CASE 1: Vertebra Exists

This case occurs when a vertebra for c_{tail} exists at N_{curr}. For example, consider appending N_2. Here, we traverse the link of N_1 to reach N_0 and find a vertebra for 'a'. Hence we create a link from N_2 to N_1 and assign it a LL of 1 (= LL of last traversed link + 1)

CASE 2: Rib With Required VL Exists

Consider appending N_4. In this case, we find that a rib for 'c' with required VL exists at N_0. Hence a link is created from N_4 to N_3 (the destination of the rib) with a LL of 1 (= LL of last link traversed + 1).

CASE 3: Rib Creation

This case occurs when there exists no rib/vertebra for the character being appended. Consider appending N_3. Traverse the link of N_2 to reach N_1. Since there exists no

rib/vertebra for 'c', make a rib from N_1 to N_3 and assign it a VL of 1. Now traverse the link of N_1 to reach N_0. And again since there exists no rib or vertebra for 'c', a rib is created for character 'c' from N_0 to N_3 with VL 0. Since the root node has no link, we end the process by creating a link from N_3 to N_0 with LL 0.

CASE 4: ExtRib Creation This case occurs when there exists a rib with VL less than the required VL. Consider appending N_7. Traverse the link of N_6 to reach N_3. At N_3, there exists a rib for character 'a' (the character being appended) but with VL less than the LL of the link last traversed ($1 < 2$). And we see that there is no extrib from N_5 (the destination node of the rib). So, an extrib is created from N_5 to N_7 and its VL and PRL are set to 2 (LL of the last traversed link) and 1 (VL of the rib), respectively. Then, a link is created from N_7 to N_5 (the last traversed rib/extrib with the same PRL as the newly created extrib) with a LL of 2 (= LL of the last traversed rib/extrib belonging to the same PRL + 1).

7.4 Implicit and True SPINE

A suffix tree can have two versions: *implicit* and *true* [1]. In the implicit tree it is possible that there are suffixes that do not terminate at a leaf node – this happens when a suffix is a prefix of another suffix in the sequence. This feature is eliminated in the true suffix tree by the simple expedient of adding a special character '$' to the end of every path in the tree, thereby ensuring that all suffixes terminate in leaf nodes.

Similar to the above, we can also have *Implicit SPINE* and *True SPINE*. The SPINE indexes discussed so far have been implicit SPINE, where all suffixes do not have to terminate at the tail node. But, just like suffix trees, by using '$' to terminate the sequence, we can generate a true SPINE that will have all suffixes terminating at the tail node.

APPEND $(n+1)^{th}$ **character**

01. c_{tail} = $(n+1)^{th}$ character
02. Append N_{n+1} to the SPINE using a vertebra
03. $N_{tail} = N_{n+1}$
04. $N_{curr} = Link(n)_{Dest}$
05. *found* = FALSE
06. WHILE (NOT *found*)
07. IF ($N_{curr} \neq NULL$)
08. s = Most recently traversed link
09. Check for a c_{tail} vertebra/rib at N_{curr}
10. IF (no edge found) **/* CASE 3 */**
11. Create a rib r from N_{curr} to N_{tail};
12. $r_{CL} = c_{tail}$;
13. $r_{VL} = s_{LL}$;
14. ELSE IF (a vertebra is found) **/* CASE 1 */**
15. $newLink_{Dest} = N_{curr+1}$;
16. $newLink_{LL} = s_{LL} + 1$;
17. found = TRUE;
18. ELSE IF (a rib r is found)
19. IF ($s_{LL} > r_{VL}$) **/* CASE 4 */**
20. Search for an extrib e with $e_{VL} >= s_{LL}$ & $e_{PRL} = r_{VL}$
21. IF (there exists such e)
22. $newLink_{Dest} = e_{Dest}$;
23. $newLink_{LL} = s_{LL} + 1$;
24. ELSE
25. Create an extrib e from the extrib chain end to N_{tail}.
26. $e_{VL} = s_{LL}$;
27. $e_{PRL} = r_{VL}$;
28. Let $lrib$ = last rib/extrib traversed with PRL = e_{PRL}.
29. $newLink_{Dest} = lrib_{Dest}$;
30. $newLink_{LL} = lrib_{VL} + 1$;
31. ELSE **/* CASE 2 */**
32. $newLink_{Dest} = r_{Dest}$;
33. $newLink_{LL} = s_{LL} + 1$;
34. found = TRUE;
35. END-IF;
36. IF (found) //Construction process ends
37. Create $newLink$ from N_{tail};
38. ELSE // Still more suffixes to be updated
39. $N_{curr} = link(n_{curr})_{Dest}$;
40. END-IF;
41. ELSE // link chain ends
42. $newLink_{Dest} = 0$;
43. $newLink_{LL} = 0$;
44. Create link $newLink$;
45. found = TRUE;
46. END-IF;
47. END-WHILE;

Figure 7.1: **SPINE Construction Algorithm**

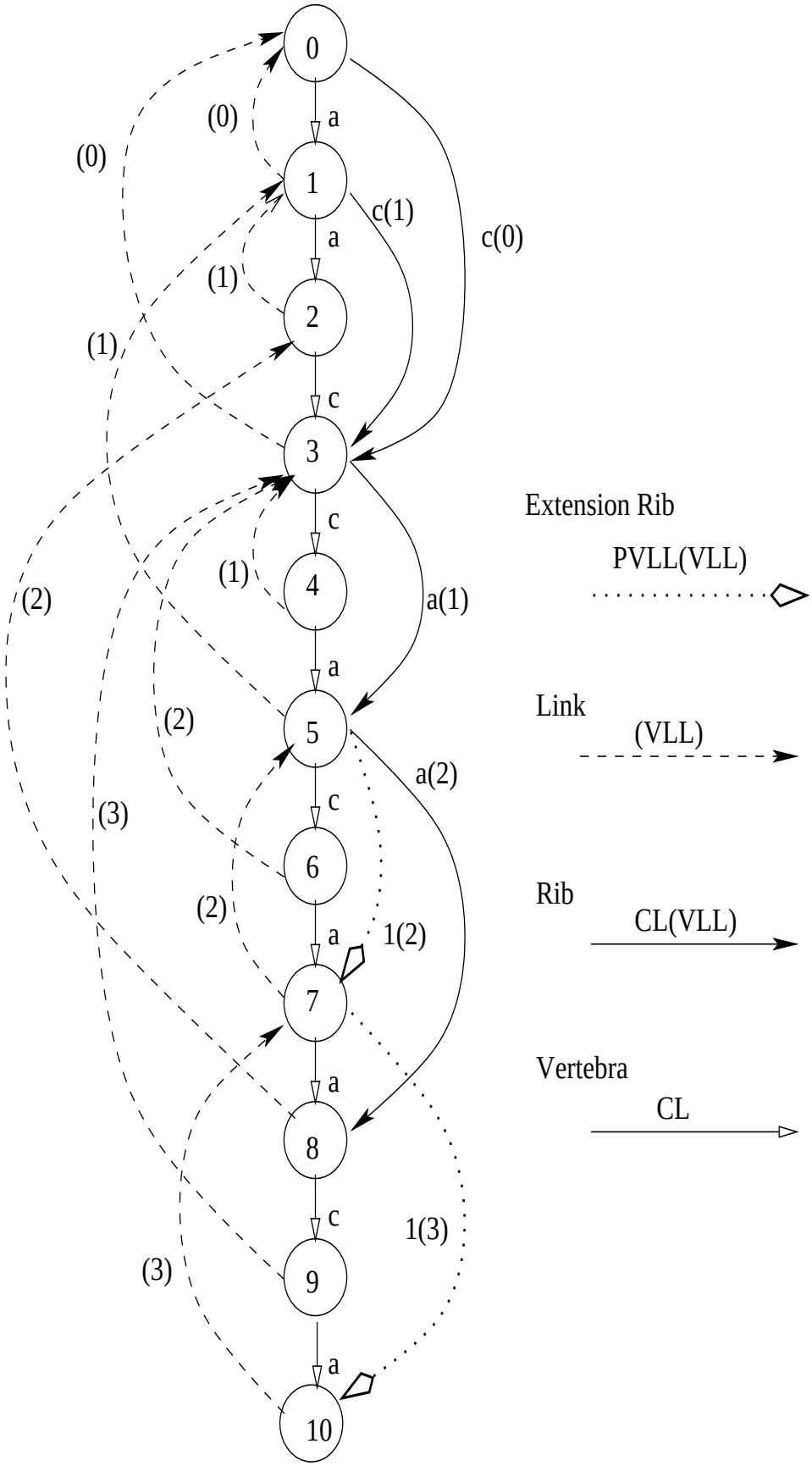


Figure 7.2: **Example SPINE Construction (for *aaccacaaca*)**

Chapter 8

Proof of Correctness

Notation

N_i - i^{th} node on the backbone

S_i - string on the backbone till N_i

$ALLSUF_i$ - set of all suffixes of S_i

$ALLSUF_i(l)$ - set of suffixes of S_i of length less than or equal to l

END_i - set of suffixes of S_i ending at N_i

$END_i(l)$ - set of suffixes in END_i of length less than or equal to l

s_{ij} - suffix of S_i of length j

LL_i - LL of the link of N_i

$DESTLINK_i$ - destination node number of the link of N_i

$DESTRIB_i^c$ - destination node number of the rib at N_i for character c

VL_i^c - VL of the rib at N_i for character c

Lemma 1:

A valid path from root to any node, say N_i, indicates that the string corresponding to the path is a suffix of S_i.

Proof:

We can reach a node through a vertebra or a rib/extension rib.

First consider the case of a vertebra. A vertebra into some node N_i always extends some suffixes of S_{i-1} to get the suffixes of S_i. Hence any path to a node N_i, which ends in a vertebra represents some suffix of S_i.

Second, let us consider when a rib is created. When appending any node N_i, we extend the suffixes of S_{i-1} to get the suffixes of S_i. Some of these suffixes, as explained in last chapters, are extended by using ribs/extension ribs during the construction process. And according to the construction algorithm, we create a rib from some node $N_j(j < i)$ if we have to extend some suffix s_{jk}(which is same as $s_{(i-1)k}$) to obtain $s_{i(k+1)}$ and the destination of all the ribs created during appending of N_i is N_i. In other words, ribs are created to accomodate the suffixes that are newly created by the addition of new character at the end of the sequence. For example, $s_{(i-1)k}$ is a suffix of both S_j and S_{i-1} but $s_{i(k+1)}$($s_{(i-1)k}$ concatenated by the newly appended character) occurs only in S_i. So, ribs coming into any node N_i, are created only when that node is being appended to the vertebra and no other time and each one of these ribs correspond to different suffixes of S_i. So a path at any node N_i which ends in a rib always represents a suffix of the S_i. The above explanation is true for extension ribs as well because the destination of an extension rib is always the node that is being appended.

Hence, any valid path from the root node to any node N_i always represents a suffix of the string S_i.

Theorem:

The addition of ribs and extension ribs do not create false positives in SPINE

Proof: Consider that we have no *false positives* in SPINE for S_i and we are extending it by some character c. Appending a character to the Spine is nothing but extending the previously existent suffixes by the newly added character and these suffixes are retrieved by traversing up the link chain. This is equivalent to extending all the suffixes of S_i by c.

Now we have to prove that no additional paths other than those corresponding to the new suffixes of S_{i+1} are created on SPINE when N_{i+1} is appended to the backbone.

When N_{i+1} is appended at the end of backbone, all the suffixes of S_i in END_i are automatically extended by the new vertebra at the end. According to Lemma 1, we have that all the paths to any node N_i represent suffixes of S_i and therefore we are extending only the valid suffixes and none other.Hence the addition of vertebra does not create any false positives.

Next we consider the additional paths created by addition of ribs/extension ribs. Now for extending the other suffixes of S_i which do not belong to END_i, we traverse the link chain starting from N_i as explained in section 2.1. Suppose from N_i, we traverse the link and reach $N_j (j = DESTLINK_i)$, and the LL_i is l. By definition of a link we have, $ALLSUF_i(l) \equiv ALLSUF_j(l)$. Now at N_j,

Case 1: No Rib/Vertebra exists

By definition of link, we have s_{il} ending at N_j. But since there exists no rib/vertebra for c at N_j, none of the suffixes in END_j has been extended by c.

In particular, $END_j(l)$ represents the suffixes in $ALLSUF_i(l)$ ending at N_j and only these suffixes and not all the suffixes in END_j, need to be extended by c.

Acoording to the construction algorithm, a new rib is created at N_j for c and given a VL of l.

This VL indicates that the rib is valid only for suffixes in $END_j(l)$ i.e only the suffixes of length less than or equal to l ending at N_j are extended.

Since we have no false positives in SPINE for S_i, we have no false positives for SPINE forS_j $(j < i)$. Now again using Lemma 1, all the paths that end at N_j are suffixes of S_j and this set is nothing but END_j. And using the VL of the rib, we ensure that only the required suffixes are extended. Hence, the addition of a rib causes *no false positives* when appeding N_{i+1}.

Now we have $END_j(l) \subseteq ALLSUF_i(l)$ and by creating rib at N_j we extended only the suffixes in $END_j(l)$. So, we have to check the other suffixes in the set $ALLSUF_i(l) - END_j(l)$

But we have $(ALLSUF_i(l) \equiv ALLSUF_j(l))$ and the suffixes in the set $ALLSUF_i(l) - END_j(l)$ can be retrieved by traversing the link chain at N_j. Therefore we traverse the link at N_j and repeat the same process.

Case 2: Extension Rib creation

Let $dest = DESTRIB_j^c$ and $vl = VL_j^c$

This case occurs when $(vl < l)$. This indicates that this rib is valid for $END_j(vl)$ but not s_{jl}. Hence we create a extension rib from N_{dest} to N_{i+1} and its given a VL of l. This VL ensures only the required suffixes are extended as explained in Case 1.

Now there can be multiple ribs ending at N_{dest} and and we need to identify to which one of them the newly created extension rib corresponds to.

We add additional information parent rib label(PRL), to the extension which is nothing but the VL of the rib which failed the test. This works because all the ribs ending at a node have different VLs because all of them are extending different suffixes of the same string. Thus the extension ribs do not create any false positives.

So, we have seen that given a SPINE for S_i, addition of ribs/extension ribs to it to obtain SPINE for S_{i+1} doesn't create any false positives.

Chapter 9

Searching Using SPINE

We now move on to describing how SPINE can be used for the various search operations that are commonly used by biologists. We first discuss how to find an exact match of a given query pattern in a data sequence. Then we extend this algorithm to find all the exactly matching subsequences between two given data sequences. Finally, we discuss how to obtain all the occurrences of a query pattern, and then describe how approximate matching can be done on SPINE.

9.1 Longest Prefix Match

Here our goal is to find the *first occurrence* of the *longest prefix* of the query pattern that is present in the data sequence. The procedure for achieving this goal on SPINE is shown as Algorithm MAX_MATCH in Figure 9.1.

In this algorithm, we start from the root node and traverse all the forward edges (vertebras, ribs and extribs) in accordance with the query pattern. A vertebra edge can be traversed at any time. Before traversing a rib, however, a check is made as to whether the length traversed thus far is *leq* VL of the rib. If this test fails, then the extrib chain is followed until we find an extrib belonging to the group of the rib which failed the test (with PRL equal to the VL of the rib) and VL $\geq$ length traversed till now.

The intuition behind our searching scheme is simple: Each valid path starting from the

root to a node corresponds to some suffix of the SPINE till that node. And as mentioned earlier, each such suffix would be of a different length. So, if it is valid to traverse a rib after a pattern p of length k, then it has to be valid after a pattern q whose length is less than k, because q would be a suffix of p.

9.2 Subsequence Matching

One of the most common operations in genome processing is *alignment*. To align two given sequences, we first need to find all the *subsequence exact matches* [1] in the two sequences. For example, in the two strings S1 and S2 shown below, the matching subsequences are shown in boldface.

S1 *acacc***gacgat***acga***gattacgaga***cgagaatacaacag*

S2 *catagagagac***gattacgaga***aaacgggaaa***gacgat***cc*

A brute-force approach to perform this alignment is to enumerate all the subsequences of the shorter sequence and match it with all the corresponding subsequences of the longer sequence. The complexity of this brute-force approach is $O(nm^2)$ subsequence matchings, where m and n are the lengths of the two data sequences, with $m < n$.

Using SPINE, this time complexity is reduced to just $O(m^2)$. The methodology used is as follows: A SPINE index is built for the longer of the two sequences. Then, the other sequence is searched for in this index. As soon as the first mismatch is found, the length matched till now is reported. Now, we check if the mismatched character follows any of the shorter suffixes of the second sequence traversed till now, and the process is repeated again. The shorter suffixes are reached by traversing the link chain upwards. So, only one check is made for each $EndSuf_i$, reducing the computational cost to $O(m^2)$.

The algorithm describing the subsequence matching is outlined in Figure 9.

[1] It is actually *substring exact matches*. In this report however, we have used the terminology 'subsequence' for 'substring'.

Comparison with Suffix Trees

Similar to SPINE's use of links, suffix trees use *suffix links* to assist in finding all the exactly matching subsequences in two given sequences. But, the number of suffixes checked by suffix tree indexes is *far more* than the ones checked by SPINE. The reason for this is as follows: In suffix trees, a suffix link points from a node indexing string aw to the node indexing w, where 'a' is a character and w is a string [1]. In the case of a mismatch, after checking for aw, we retrieve the node indexing the suffix w and check if the mismatched character follows w. This process iterates till complete match is found or there are no more suffixes remaining to be checked.

In contrast, as mentioned earlier, in SPINE each node N_i in a link chain represents a *set* of suffixes $EndSuf_i$. Therefore, only one check is sufficient for all the suffixes in that set. Therefore, lesser number of suffixes are checked, thereby reducing the computational effort.

This would be clear from the Figure 9.3 Suppose in a suffix tree, a mismatch is found below node 2, then the next suffix to be checked will be below node 4 (node pointed to by the suffix link), which will give a suffix of length one lesser. On the other hand, in SPINE, the link points to node 1 which actually represents the suffixes of length 1 and less i.e 3 lesser than the suffixes at node 4. And therefore, for large sequnces, a very small number of suffixes are actually checked if SPINE index is used.

9.3 Multiple Occurences of a Pattern

Here our goal is to find *all* occurrences of the query string in the data sequence. This is achieved using a simple technique with the help of links. The property of the links exploited here is that a link with LL v from node N_a to node N_b indicates that a sequence of length v above N_a is same as the sequence of length v above N_b.

We start off with finding the first occurrence of the given pattern, as per the algorithm given in the last section. The node indexing the first occurrence is stored in a *target node buffer*. Then all the nodes downstream are scanned successively to check if their links

point to the node in the target node buffer, i.e. the node indexing the first occurrence. If so, then that node is also stored in the target node buffer. Again, downstream scanning is started from this node and the process is repeated until the end of the backbone is reached.

For a sequence of length n and search pattern of length m, if the number of occurrences of pattern in the sequence is p, then the size of the target node buffer will be p integers. And at each node, the target node buffer will be searched in binary fashion. So, the time required to locate all the occurrences is $O(n\ log\ p)$.

To clarify the above, consider Figure 7.2 with a query sequence `ac`. Here, after locating the first occurrence, the target node buffer will contain N_3. Moving downstream, at N_6 we find a link with LL = 2 (length of string `ac`) pointing to N_3. And so, N_6 is also added to the target node buffer. On moving further downstream, at N_9, a link with appropriate LL is found pointing to a node in the target node buffer (N_3), and therefore it is also added to the buffer. In this manner, the target node buffer finally gives the *end* nodes of all occurrences of the pattern in the sequence. As a last step, their starting positions can be trivially determined by merely subtracting the query pattern length from each of the node identifiers in the target node buffer.

In case of finding all the *subsequence* matches, the above technique amortizes the cost. This is due to the fact that to find all the occurrences of all the subsequences, only a single final sequential scan of the complete backbone is required, which incurs a negligible cost as compared to the total cost of finding the first occurrences of the matches.

9.4 Multiple Base Sequences

Sometimes, the pattern is required to be searched for in more than one sequences. Instead of building a separate SPINE for each sequence and doing the search, a single index structure can be made for all of them. A pattern being searched for cannot occur across the strings, i.e. a part of it in one string and remaining in another. But the same pattern may occur completely in more than one string. To report all the occurrences in all the strings, a small enhancement in the methodology discussed in the last section could be

done. A delimiter is used after every sequence, say '@'. So between any two contiguous sequences, there will be a '@'. So, search for any pattern, not including '@', cannot result into partial occurrences of the pattern in any string. And using the links with the methodology to find the multiple occurences, all the occurrences can be reported.

9.5 Approximate Matching

The goal in *approximate* sequence matching is to find patterns in the data sequence that are within an *edit-distance* threshold of the query pattern (edit-distance is the minimum number of insertions or deletions that have to be made to transform one sequence into another). Typically, dynamic programming is used for efficient implementation of this operation [1]. To find the best approximate matches, the edit distance is calculated for all the suffixes with the query pattern sequence. Suffix trees aid in this method by providing all the possible suffixes, reducing the redundant computations [1].

The above method can be employed equally well over SPINE, since it also provides all the valid suffixes by traversing the ribs, extribs and vertebras in a depth-first manner. So, the edit distances could be calculated easily by traversing each path, one at a time, and finding the edit distance with respect to the query sequence. And according to the obtained edit distances, queries could be answered.

It has been shown that *suffix links* in a suffix tree can be used to optimize the above techniques [1]. These optimizations can all be applied as such in the case of SPINE also by using its *link* edges.

```
MAX_MATCH
01. travLen = 0; // Length traversed till now
02. curr = 0; // Current Node Number
03. mismatchFound = FALSE;
04. WHILE ((NOT end of query) and
    (NOT mismatchFound))
05.     ch = queryseq.nextchar();
06.     IF ch is seqchar of N_curr
07.         travLen++;
08.         curr++;
09.     ELSEIF there exists a rib R for ch at N_curr
10.         IF R.VL >= travLen
11.             travLen++;
12.             curr = R.Dest;
13.         ELSE
14.             PRL = R.VL;
15.             temp = R.Dest;
16.             extribFound = FALSE;
17.             WHILE ((NOT extribFound) and (extrib exists at N_temp))
18.                 IF extrib.PRL = PRL
19.                     IF extrib.VL >= travLen
20.                         extribFound = TRUE;
21.                         travLen++;
22.                         curr = extrib.Dest;
23.                     ELSE
24.                         temp = extrib.Dest;
25.                     END-IF
26.                 END-IF
27.             END-WHILE
28.             IF ( NOT extribFound )
29.                 mismatchFound = TRUE;
30.             END-IF
31.         END-IF
32.     ELSEIF
33.         mismatchFound = TRUE;
34.     END-IF
35. END-WHILE
36. matchStartPosition = curr - travLen + 1;
    //matchStartPosition gives the starting position of the longest matching prefix
```

Figure 9.1: **Longest Prefix Match**

SUBSEQUENCE_MATCH

// m is the length of s, the shorter sequence

01. $curr = 0$;
02. $pos = 0$;
03. WHILE (NOT ENDOF(s))
04. Start MAX_MATCH from N_{curr} for $s[pos..m]$
05. Report the maximum match
06. pos = position of mismatched character in the query;
07. $curr = link(curr).DEST$;
08. END-WHILE

Figure 9.2: **Subsequence Matching**

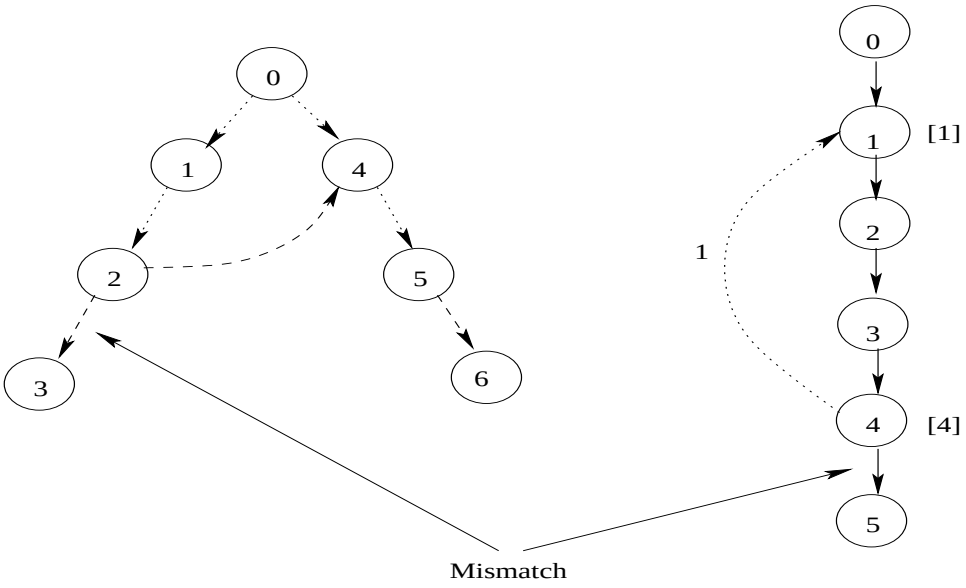


Figure 9.3: **Subsequence Matching (SPINE v/s Suffix Tree)**

Chapter 10

Implementation Details

We have developed a prototype version of SPINE, and in this chapter, we discuss its implementation details.

All the nodes, one per character in the sequence, are stored sequentially and therefore, the edge between adjacent nodes is *implicit*. Apart from the CL of the vertebra, Table 10.1 gives all the information fields required at each node and also the space requirements associated with them.

Field Name	**Space (Bytes)**	**Count**	**Total (Bytes)**
Link Dest	4	1	4
Link LL	4	1	4
Rib Dest	4	3	12
Rib VL	4	3	12
ExtRib Dest	4	1	4
ExtRib VL	4	1	4
ExtRib PRL	4	1	4

Table 10.1: **Node Space Requirement (44 bytes)**

10.1 Node Size Optimizations

As can be seen from Table 10.1, the space required by each SPINE node is huge (44 bytes) with a straightforward implementation. However, SPINE exhibits certain intrinsic

features using which the actual space required can be reduced to a great extent. A variety of space-reducing optimizations based on these features are outlined below:

10.1.1 Optimization 1: Small Label Values

Table 10.2 gives the maximum label value observed for a variety of real and synthetic genome sequences – as can be observed here, the label values never exceed 25000 even for very large sequences. Therefore, only two bytes need to be allocated for the length fields.

Genome Sequence	MaxLabel
E.Coli (3.5Mbp)	1785
C.Elegans (15.5 Mbp)	8187
HG chr21 (28.5 Mbp)	21844
Synthetic DNA (40 Mbp)	33
HG chr19 (57.5 Mbp)	12371
Synthetic Protein (1 Mbp)	23
Synthetic Protein (10 Mbp)	28

Table 10.2: **Maximum Labels Observed**

For synthetic generated pseudorandom sequences, the above empirical observation can also be justified as follows: A rib with a VL of 2^{16} is created only when an entire sequence of 2^{16} characters repeat. Given a sequence of length 2^{16}, the probability that we have the same sequence again is $1/(A^{2^{16}})$, which is extremely low. Here A represents the alphabet size (4 for DNA sequences and 20 for protein sequences).

However, to ensure that the index works in all cases, we have a mechanism in place to handle even those rare cases where the label value goes beyond 25000. We allocate separate entries for these cases in an *overflow table*. The node space for the label is used to index into the overflow table, and a one bit flag is used to indicate whether it is a valid label or an index into the overflow table. If that bit is set, then the label field indicates the pointer in the overflow table.

Maximum Label Values are independent of Sequence Length

From Figure 10.1, it can be observed that the maximum label values obtained are a property of the patterns in the sequence and not its length. For instance, after constructing the same length for both C Elegans and 19th choromose of HG, the maximum label values are found different.

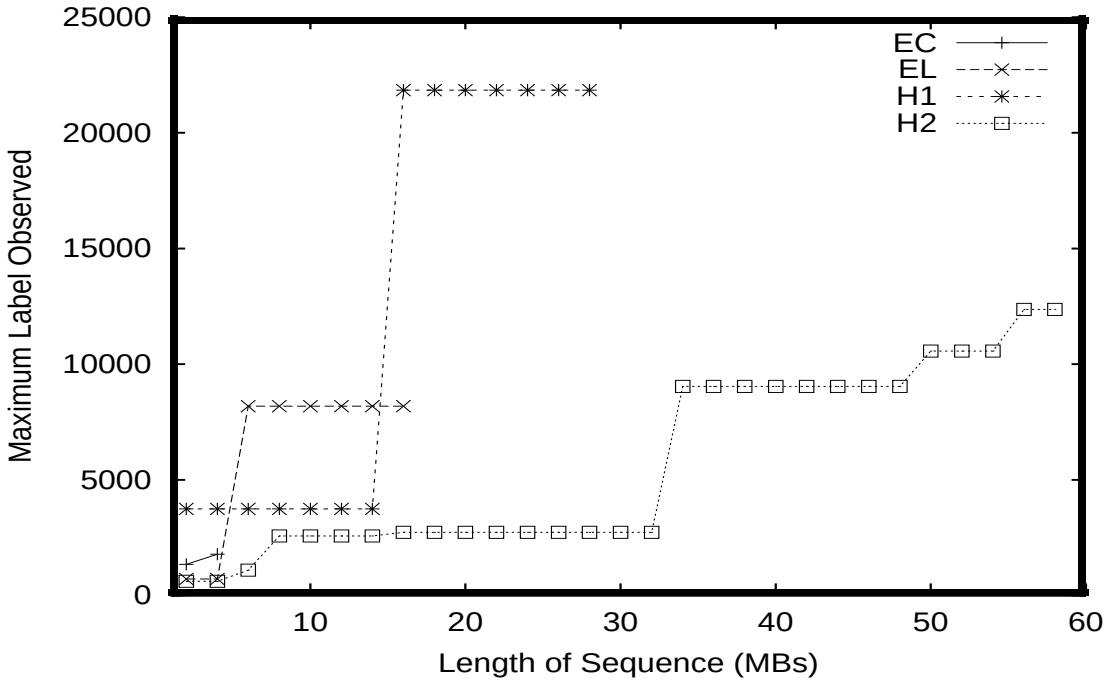


Figure 10.1: **Growth in Maximum Label Value**

10.1.2 Optimization 2: Sparse Rib Distribution

While all nodes have upstream edges (links), the same is not true with respect to downstream edges (ribs and extribs i.e. extension ribs). In fact, we have found that only around *30 to 35 percent* of the nodes actually have any downstream edges emanating from them – Table 10.3 shows the distribution of their number across the nodes for the various DNA sequences, while Figure 10.2 shows the same for protein sequences. The reason for this behavior is that after some length of the sequence, the remaining part merely contains repetitions of previously occurred patterns, and therefore not many downstream edges are created.

Sequence	1	2	3	4	Total
E Coli	15	9	6	4	33
C Elegans	15	8	6	4	33
HG chr21	14	8	6	4	32
Synthetic	15	9	6	4	34
HG chr19	13	7	5	3	28

Table 10.3: **Rib Distribution across Nodes in Nucleotides (in percent)**

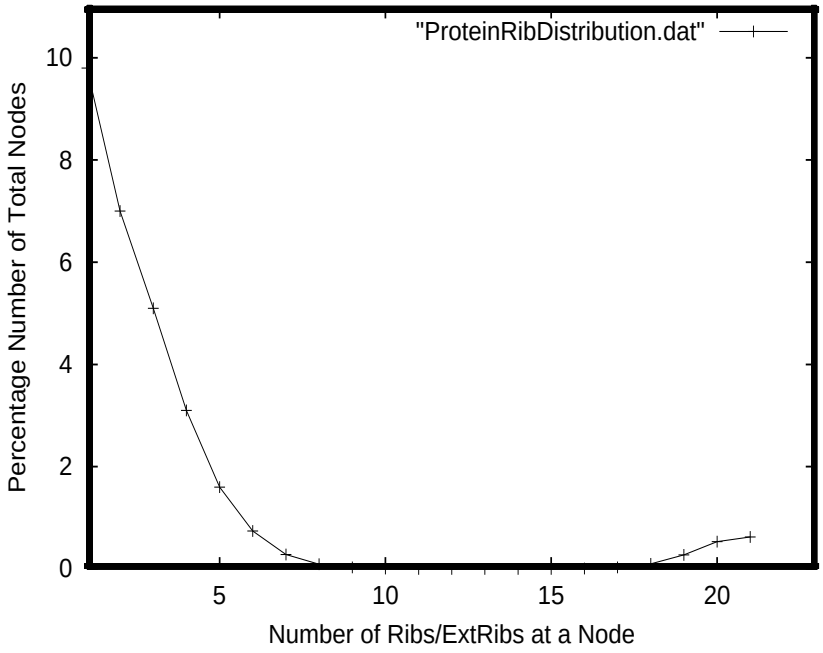


Figure 10.2: **Rib Distribution across Nodes in Proteins**

Based on the above observation, we do not allocate space for downstream edges at every node, since a lot of space would be wasted. Instead, we store information about the links and the downstream edges separately in a *Link Table* (LT) and a *Downstream Edge Table* (DT), respectively. One entry for each character in the sequence is allocated space statically in the *LT*, while space for downstream edges is allocated dynamically in the *DT* for only those nodes from which a rib/extrib emanates. Therefore, the total number of entries in the DT is only around 30 to 35 percent of that in the LT.

10.1.3 Optimization 3: Limited Rib Fanout

From Table 10.3 it is clear that the number of nodes with a given rib fanout decreases with the fanout value. For example, only about 4% of the nodes have the full complement of all four downstream edges (three ribs and one extrib). Therefore, to avoid the space wasted for the edges which are not present, we use *multiple* DTs. Specifically, there is one DT for each possible fanout, resulting in four DTs in total: *DT1, DT2, DT3, DT4.*

This optimization results in considerable space savings. At first glance, it might appear that the construction time performance of SPINE would degrade due to the movement of nodes across the DTs, which would occur whenever a node acquires an additional downstream edge. However, we have experimentally observed that this impact is negligible.

10.1.4 Final Optimized Implementation

Based on the above discussion, the optimized implementation of SPINE consists of a **Link Table** (LT) and four **Downstream Edge Tables** (DTs), whose entries are shown in Figure 10.3. The *LT* contains one entry for each node (character) in the sequence. It stores its LL (the *V* field) as one of its columns while the other column represents either the destination node of that link (the LD field) or a pointer to an entry in one of the DTs (the PTR field). In particular, the *LT* stores the link destinations only for the nodes that don't have any ribs/extrib. For the remaining nodes, they are stored in the DT entries only.

Typically, a DT entry stores the destination node of the link from that node and also the destination nodes and VLs of all the ribs/extribs starting from the same node. The *D* fields indicate the destination nodes of the downstream edges while the *V* fields denote their VLs. And, lastly, the *P* field denotes the PRL of the extrib.

By implementing all the above optimizations, the net effect is that the average node size in SPINE is *less than 12 bytes* for both DNA and protein sequences, that is, the index takes upto 12 bytes per indexed character, which is about 30 percent lesser than the space required by MUMmer.

Also, due to the comparatively smaller size of the index, the access times also decrease

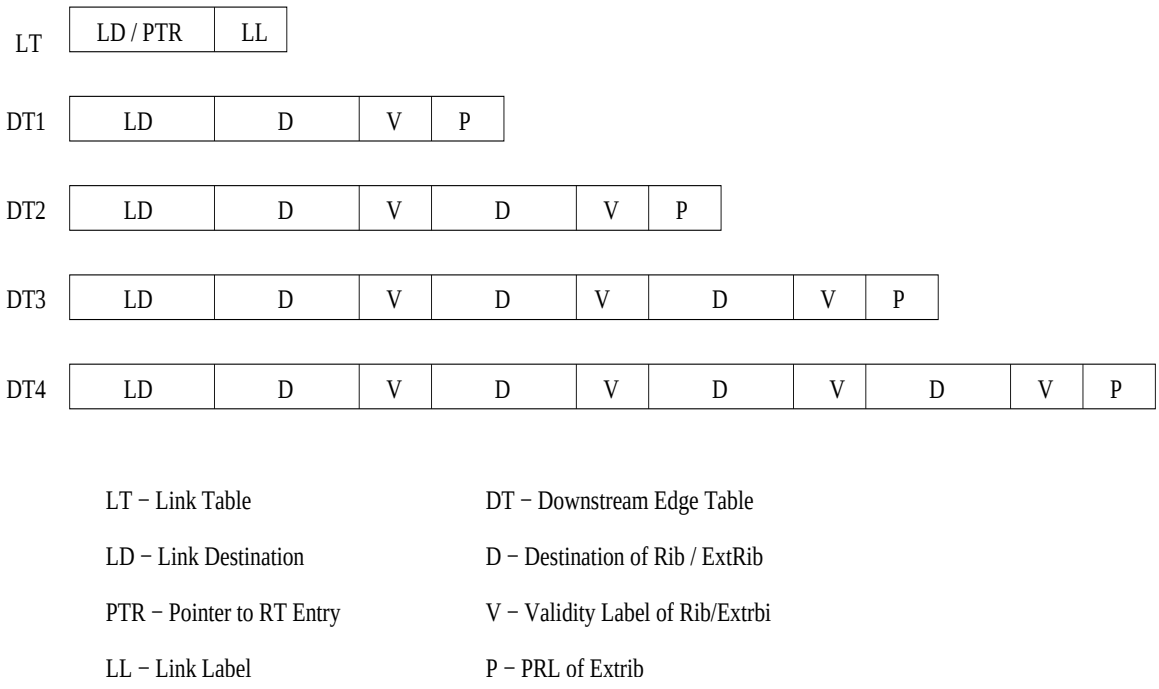


Figure 10.3: **Optimized SPINE Implementation**

influencing both the construction and searching times.

The only difference which would be reflected as a result of the increased alphabet size is that we would be able to index a comparitively smaller length sequence. This is due to the fact that the number of bits required to encode the character labels would increase with the increase in alphabet size, thereby decreasing the total length of sequence which could be indexed. But this effect is observed in most of the index structures including the suffix trees.

Chapter 11

Experimental Analysis

We conducted a detailed evaluation of the performance of the SPINE prototype, and these results are presented in this chapter. Both real as well as synthetically generated sequences were used in the experiments. The real DNA sequences used are 3.5Mbp of E. Coli (EC), 15.5 Mbp of C. Elegans (EL), 28Mbp of Chromosome 21 (H1) and 57 Mbp of Chromosome 19 (H2) of Human Genome. We also synthetically generated 40Mbp of pseuodrandom sequence (PR). For proteins, we generated synthetic sequences of length 1Mbp, 2Mbp, 5Mbp, 10Mbp and 15Mbp. The motivation behind experimenting with protein sequences also was to study the effect of increased alphabet size and also the nature of sequences themselves on the index structure.

In order to serve as comparative yardsticks, we also evaluated the performance of MUMmer representing the suffix-tree-based techniques, and BLAST (BLASTN for DNA sequences and BLASTP for protein sequences) as a representative of the seed-based methods. [1] The core data structure of MUMmer is the suffix tree, and it is implemented very efficiently using McCreight's algorithm [1]. At present, MUMmer has provided support only for DNA sequences.

BLAST (Basic Local Alignment Search Tool), on the other hand, is a set of similarity search programs designed to explore all of the available sequence databases. BLAST uses a heuristic algorithm which seeks local alignments. The alignment process done by

[1] Specifically, MUMmer Version 2.1, and BLAST Version 2.0.9 were used

BLAST can be broadly classified into three phases: *locating all the matching subsequences*, *filtering* out some regions using various scoring schemes, and finally doing the *alignment*. For our purposes we used only the first phase of the module.

All the experiments were conducted on a Pentium IV 2.4 GHz machine with 1 GB RAM, 40GB IDE disk and running the Redhat Linux 7.3 operating system. The performance metrics in the experiments were the overall time taken to build the complete index for a sequence in memory and time taken to perform searches for exact matches and all subsequence matches.

In this chapter, we will consider scenarios wherein both the data and the index structure are completely memory-resident, while disk-resident indexes are analyzed in the following chapter.

11.1 Index Construction Time

The performances of MUMmer and SPINE with regard to index construction times for nucleotides are shown in Figure 11.1. These results show that SPINE takes lesser time to construct, especially for longer sequences. Further, note that no results have been mentioned for MUMmer to construct an index for the H2 sequence as it ran out of memory due to its larger space requirements.

As mentioned earlier, MUMmer does not have support for protein sequences. And therefore, Table 11.1 gives the index construction times only for SPINE. It can be observed from the table that the construction times for proteins also scale linearly with the sequence lengths.

Protein Length	**Time (secs)**
1 Mbps	1
2 Mbps	3
5 Mbps	7
10 Mbps	14
15 Mbps	20

Table 11.1: **SPINE Construction Times for Proteins**

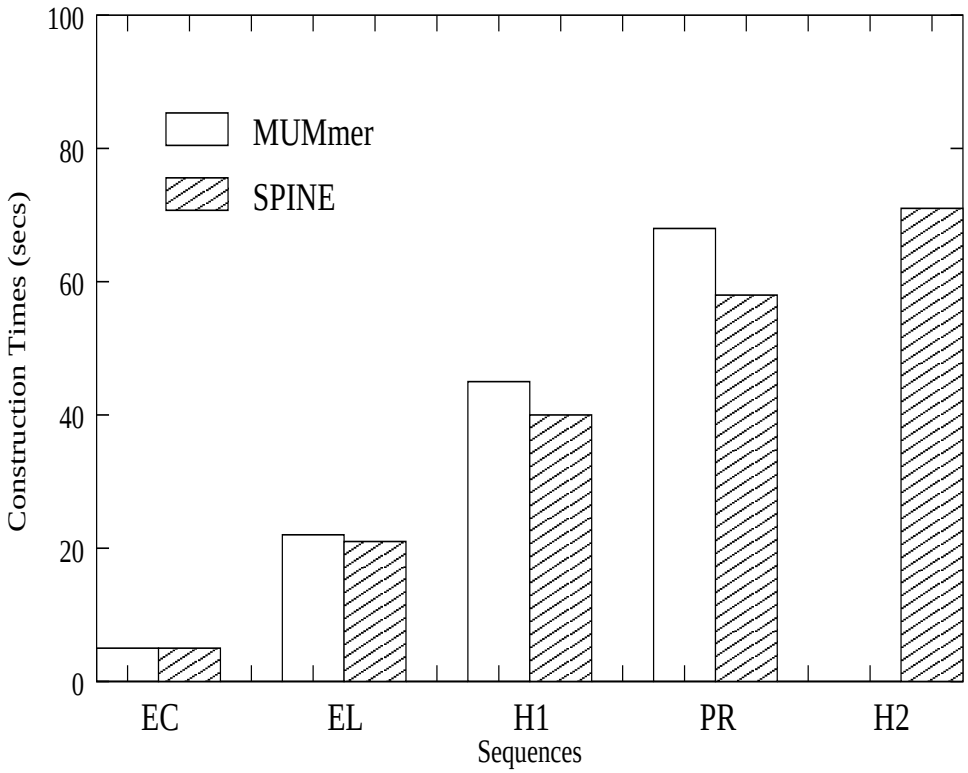


Figure 11.1: **Index Construction Times**

11.2 Search Operation Times

We now move on to comparing SPINE with MUMmer and BLAST on the search performance metric.

Longest Prefix Match

We found that the SPINE search times for this match are very small (in order of milliseconds). We could not compare them with MUMmer or BLAST as both of them do not support this match. And therefore, we compared it with the *egrep* utility and the performance comparison has been shown in Table 11.2. The pattern lengths are limited to 10K only, as *egrep* does not support patterns with the length of order of 100K and 1M.

Here we can observe that the search time is independent of both base sequence length and the query sequence length. It basically depends on the length of pattern which is actually found in the base sequence.

Seq1	Seq2	egrep	SPINE
EC	1 K	1s	0.015ms
EC	10 K	3s	0.013ms
EL	1 K	1s	0.014ms
EL	10 K	3s	0.015ms
H1	1 K	1s	0.016ms
H1	10 K	4s	0.012ms
H2	1 K	1s	0.018ms
H2	10 K	3s	0.019ms

Table 11.2: **Longest Prefix Match**

Subsequence Match

Table 11.3 gives the time required to find all the exactly matching subsequences (including all the multiple occurrences as well) for two given DNA sequences, *Seq1* and *Seq2*, for the SPINE, MUMmer and BLASTN algorithms. As expected, due to their $O(m^2)$ complexity, we find here that SPINE and MUMmer perform far better than BLASTN, especially with increasing sequence length. Specifically, SPINE is almost 200 times faster than BLASTN for E.coli, the shortest sequence, and this improvement increases to a factor of around 7000 times for Chromosome 19 of the Human Genome. Further, we also observe that SPINE takes around 30 percent lesser time than MUMmer due to handling lesser number of suffixes, as described in Section 3.2.

Seq1	Seq2 Length	BLASTN	MUMmer	SPINE
EC	1 K	217 ms	1.416 ms	1.029 ms
EC	10 K	1863 ms	13.603 ms	10.99 ms
EC	100K	20.597 s	135.697 ms	109.74 ms
EC	1 M	8m,53s	1s,359 ms	1s,70 ms
EL	1 K	991ms	1.65 ms	1.18 ms
EL	10 K	8s,549ms	16.255 ms	11.24 ms
EL	100 K	1m,30s	161.2ms	112.68 ms
EL	1 M	39m,45s	1s,622ms	1s,133 ms
H1	1 K	1803ms	1.741ms	1.198 ms
H1	10 K	20s,45ms	17.523ms	11.97 ms
H1	100 K	3m,58s	179.12ms	121.8 ms
H1	1 M	98m,12s	1s,800ms	1s,207ms
H2	1 K	3s,645ms	-	1.314ms
H2	10 K	32s,805ms	-	13.26 ms
H2	100 K	6m,20s	-	131.12 ms
H2	1 M	157m,58s	-	1s,318 ms

Table 11.3: **Subsequence Matching Time (DNAs)**

Table 11.4 gives the time required to find the subsequence matches for protein sequences and it can observed that the results are similar to those of DNA sequences. The comparison of SPINE has been done with BLASTP only, as the MUMmer does not have routines for proteins.

Seq1 Length	Seq2 Length	BLASTP	SPINE
1 M	1 K	210s	1ms
1 M	100K	22.7s	100.4ms
2 M	1 K	1001ms	1.38ms
2 M	100 K	1m,27s	115.9 ms
5 M	1 K	1783ms	1.8ms
5 M	100 K	4m,19s	125.8 ms
10 M	1 K	3s,851ms	1.800ms
10 M	100 K	6m,10s	142.8 ms

Table 11.4: **Subsequence Matching Time (Proteins)**

SPINE: An alternative to BLAST

We have experimentally observed that it is the first phase of BLAST i.e. finding all the exactly matching subsequences, which is the main bottleneck, contributing more than 95 percent of the total computation cost.

For two given sequences, *Seq1* and *Seq2*, Table 11.5 shows the portions of the total time consumed in the first phase and the remaining part of BLASTN.

Seq1	Seq2	Phase 1	Phase 2-3 together
EC	1 K	217 ms	220 ms
EC	1 M	8m,52s,844ms	9m,2s,964ms
EL	1 K	991ms	1s,18ms
EL	1 M	39m,45s,61ms	40m,30s,987ms
H1	1 K	1803ms	1841 ms
H1	1 M	98m,12s,126ms	63m,0s,23ms
H2	1 K	3.645s	3.800s
H2	1 M	157m,58s,11ms	58m,0s,138 ms
PR	1 K	2.512s	2.600 ms
PR	1 M	128m,12s,112ms	49m,0s,2 ms

Table 11.5: **BLASTN phases time distribution**

11.3 SPINE against Suffix Trees

From the last sections, we observed that there is no trade off with respect to time or space, as SPINE performs better than the suffix trees at both ends. So, we can conclude that suffix trees have some redundant information and hence horizontal compression proves to more effective than vertical compression to remove the redundancy in the tries.

Chapter 12

SPINE on Disk

We now move on to assessing the performance of a disk-resident SPINE index. While SPINE's improved performance over MUMmer in the memory-resident case could be attributed to its compactness, note that index performance on disk does not depend solely on size, but also on the *locality* of the information. To assess this performance, in Figure 12.1 we show the comparative amount of disk I/O incurred by MUMmer and SPINE for the various genomic sequences.

From these results, we see that SPINE takes only about 30% of the disk accesses as compared to MUMmer. Since the smaller node size would have reduced the disk accesses to only 70% of MUMmer, we can conclude that the additional 40% improvement is due to the improved access locality exhibited by SPINE.

Link Destination Distribution

An interesting feature that we observed about SPINE indexes is that most of the links point to the *upper* nodes in the backbone, and that the number of links pointing to a node keeps monotonically decreasing as we descend the backbone. This is shown quantitatively in Figure 12.2, which shows the distribution of the link destinations for different data sequence lengths. This indicates that while constructing the SPINE, the upstream nodes would be accessed more than the downstream ones. And so this suggests a very *simple buffering strategy* for SPINE: *"Retain as much as possible of the top part of the Link Table*

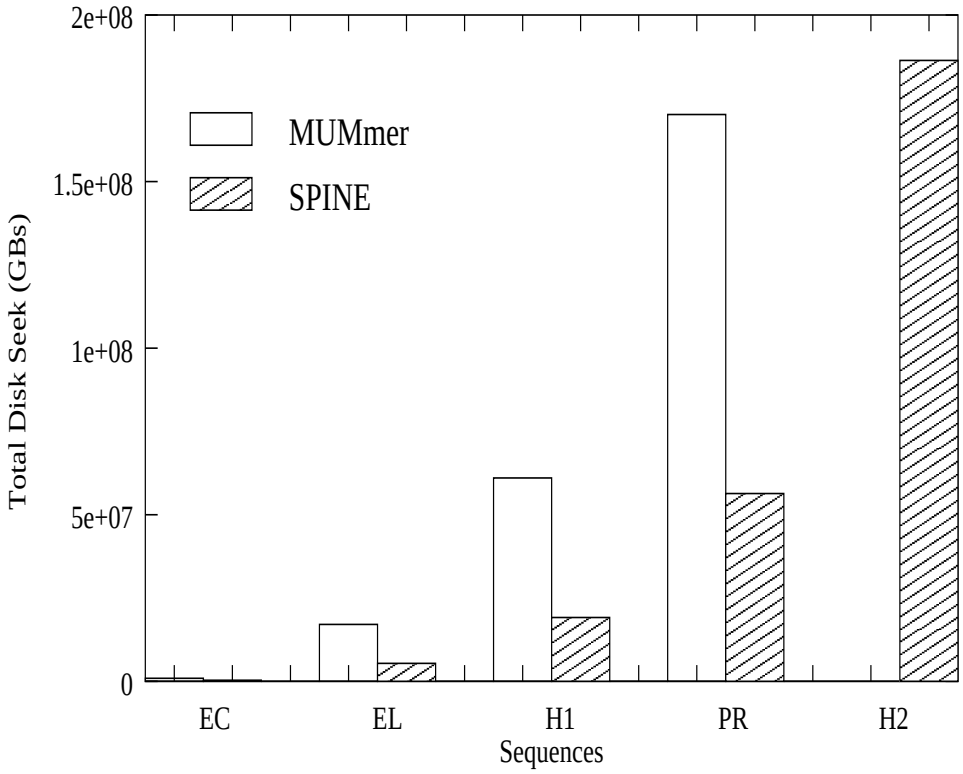


Figure 12.1: **Total Disk Seek**

in memory".

Apart from being simple, the other advantage of this buffering strategy is that no dynamic movement of information is done between the buffer and disk. So, this corresponds to a *static buffering.* In contrast, no effective buffering strategies for suffix trees have been suggested till now which may improve its performance significantly on disk.

12.1 Experimental Analysis

We performed experiments to find the improvements in the performance of SPINE over MUMmer when implemented on disk. Again, we used the index construction times and subsequence matching times as the comparison metrics.

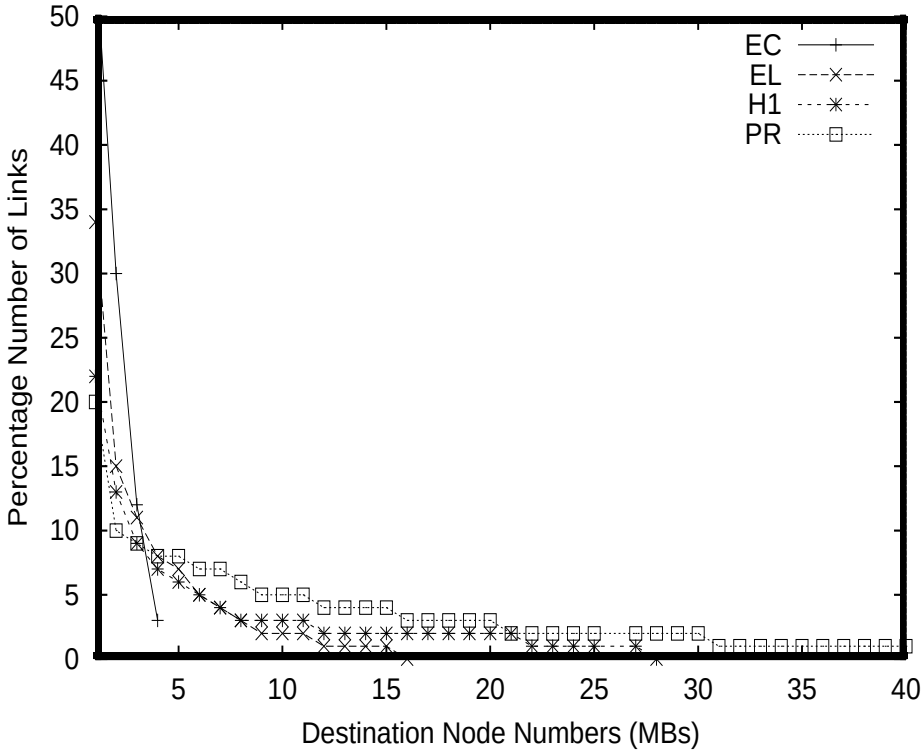


Figure 12.2: **Link Destination Distribution**

12.1.1 Index Construction Times

SPINE and MUMmer were both constructed with zero memory using synchronous I/O (O_SYNC option). With this option, absolutely no buffering is done by the operating system, and hence each read/write is done synchronously, allowing us to compare the locality behavior of the algorithms without modulation by other system factors.

The results of this experiment are shown in Figure 12.3. We see here that PINE takes almost half the time as required by MUMmer to construct the index on disk. Note that this is better than the improvement obtained for the in-memory building of the indexes, and can again be attributed due to the better spatial locality exhibited by SPINE.

12.1.2 Searching

Similar to the construction times, we observed that time required to obtain all the exactly matching subsequences also improved by a factor of two with SPINE as compared to

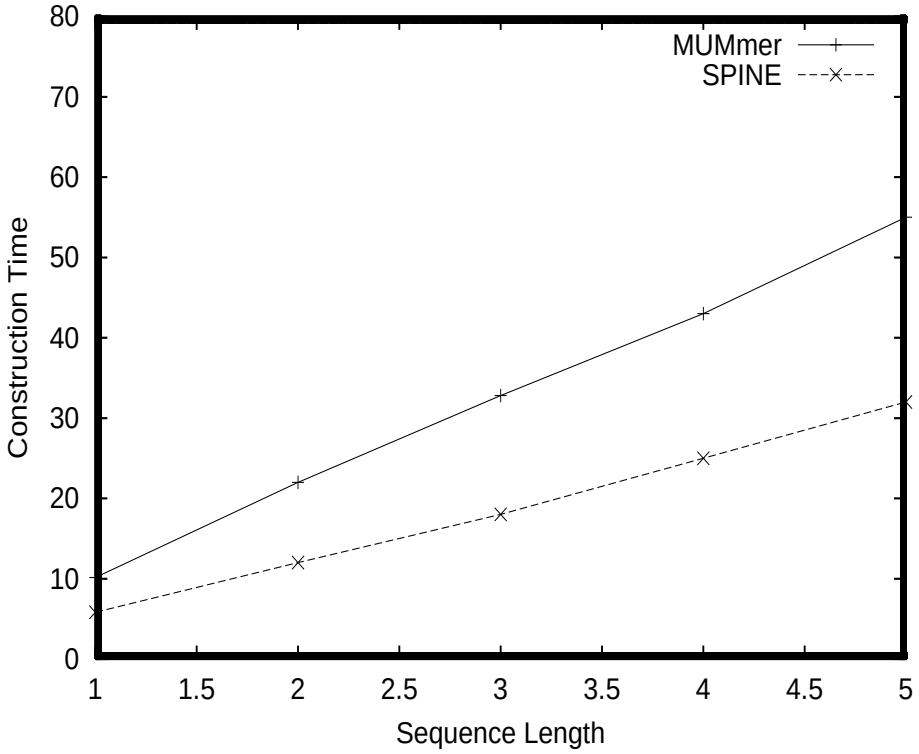


Figure 12.3: **Construction Times (Sync I/O)**

MUMmer.

In Table 12.1, we compare the time taken by SPINE and MUMmer for finding all the exactly matching subsequences for two given sequences. Again, the experimentation was done using synchronous I/O. We see from the Table that SPINE is faster by almost a factor of two.

Sequence 1 Length	Sequence 2 Length	MUMmer	SPINE
1 MB	1 K	61	33
1 MB	2 K	121	68
1 MB	4 K	242	137
5 MB	1 K	59	31
5 MB	2 K	120	67
5 MB	4 K	250	134

Table 12.1: **Subsequence Matching Times (secs)**

Chapter 13

Conclusion

We have proposed the SPINE data structure for biological sequence indexing. SPINE implements a complete horizontal compaction of the basic trie structure, ensuring that each character in the data sequence is represented only once in the index, and thereby reduces the space requirements to a great extent as compared to the traditional suffix trees. The false positives that resulted from this compaction were removed through a rib and link labeling strategy that constrains when these graph edges can be traversed. An online construction algorithm for SPINE has been explained in detail. The viability of a parallel construction algorithm for SPINE needs to be explored.

While a simplistic implementation of SPINE would have resulted in huge node sizes, we identified and incorporated a variety of structural optimizations that finally resulted in SPINE taking less than twelve bytes per character indexed, comparing favorably with the 18 bytes taken by suffix trees.

Also, it was shown that all types of search operations required by biologists to be performed on genome sequences can be done using SPINE, making it a versatile index structure.

A performance evaluation of SPINE against MUMmer and BLAST over various real DNA sequences showed that significant speedups were obtained for the searching operations, for both memory-resident and disk-resident scenarios. In fact, SPINE performed close to four orders of magnitude faster than BLAST. It was also observed that along

with 30 percent lesser index size, SPINE exhibits much higher node locality than MUMmer, resulting in a more efficient disk-based implementation. Table 13.1 summarizes the normalized comparison results obtained for SPINE against MUMmer.

Criterion	MUMmer	SPINE
Space Requirements	1.5	1
Index Construction Times (Memory)	More	Less
Subsequence Matching Times (Memory)	1.5	1
Total Disk Seek	3	1
Index Construction Times (Disk)	2	1
Subsequence Matching Times (Disk)	2	1
Buffering Strategy	Complex	Simple

Table 13.1: **Index Structures - MUMmer v/s SPINE**

Bibliography

[1] D. Gusfield, "Algorithms on Strings, Trees, and Sequences", *Cambridge University Press*, 1997.

[2] E. McCreight, "A space-economical suffix tree construction algorithm", *J. Algorithms*, 23(2):262-272, 1976.

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

[4] R. Giegerich, S. Kurtz and Jens Stoye, "Efficient Implementation of Lazy Suffix Trees", *Proceedings of 3rd Workshop On Algorithm Engineering, London, UK*, July 1999.

[5] M. Crochmore and R. Verin, "Direct Construction of Compact Acyclic Word Graphs", *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, pages 116-129, 1997.

[6] A. Delcher et al., "Alignment of whole genomes", *Nucleic Acids Research*, 27:2369-2376, 1999.

[7] S. Altschul and W. Gish, "Basic Local Alignment Search Tool", *J. Mol. Biol.*, 215, 403-410, 1990.

[8] S. Inenaga et al., "On-Line Construction of Compact Directed Acyclic Word Graphs".

[9] U. Manber and G. Myers, "Suffix arrays : a new method for on-line string searches", *SIAM J. Comput.*, 22(5):935-948, 1993.

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

[11] T. Kahveci and Ambuj K. Singh, "An Efficient Index Structure for String Databases", *Proc. of the 27th VLDB Conference, Roma, Italy*, 2001.

[12] T. Kahveci and Ambuj K. Singh, "MAP: Searching Large Genome Databases", *Pacific Symposium on Biocomputing, Kaua'i, Hawaii*, 2003.

[13] B. Ma, J. Tromp and M. Li, "Pattern Hunter: Faster And More Sensitive Homology Search", *Bioinformatics*, 18(3):440-445, 2002.

[14] *http://www.ncbi.nlm.nih.gov*

[15] *http://www.ebi.ac.uk/embl*

[16] *http://www.tigr.org/software/mummer*

[17] *http://www.expasy.ch/sprot*