A HORIZONTALLY-COMPACTED TRIE INDEX FOR STRINGS

Naresh Neelapala Romil Mittal Jayant R. Haritsa

Technical Report
TR-2003-05

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

A Horizontally-Compacted Trie Index for Strings

Naresh Neelapala Romil Mittal Jayant R. Haritsa
Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, INDIA
{naresh,romil,haritsa}@dsl.serc.iisc.ernet.in

Abstract

The indexing technique commonly used for long strings, such as genomes, is the suffix tree, which is based on a vertical (intra-path) compaction of the underlying trie structure. In this paper, we investigate an alternative approach to index building, based on horizontal (inter-path) compaction of the trie. In particular, we present SPINE, a carefully engineered horizontally-compacted trie index. SPINE consists of a backbone formed by a linear chain of nodes representing the underlying string, with the nodes connected by a rich set of edges for facilitating fast forward and backward traversals over the backbone during index construction and query search. A special feature of SPINE is that it collapses the trie into a linear structure, representing the logical extreme of horizontal compaction.

We describe algorithms for SPINE construction and for searching this index to find the occurrences of query patterns. Our experimental results on a variety of real genomic and proteomic strings show that SPINE requires significantly less space than standard implementations of suffix trees. Further, SPINE takes lesser time for both construction and search as compared to suffix trees, especially when the index is disk-resident. Finally, the linearity of its structure makes it more amenable for integration with database engines.

1. Introduction

A wide variety of applications require searching for exact or approximate matches over long text strings [1]. For example, performing global alignment between a pair of genomes that each run to millions or billions of nucleotides is a common task undertaken by biologists, the core operation of which is searching for maximal unique matches across the genomic strings [5]. Since brute-force searching techniques do not scale to such long strings, there has been extensive research on the design of high-performance *index structures* for strings.

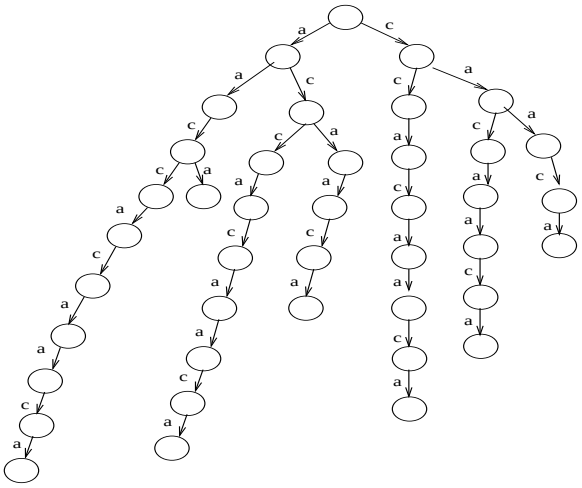

Figure 1. TRIE (for *aaccacaaca*)

In this context, a trie [13] which holds all suffixes of the data string, has become a popular starting point for developing index structures [1]. An example trie for the string aaccacaaca is shown in Figure 1. A space-efficient version of the trie structure, called the *suffix tree*, can be created by collapsing every unary node of the trie into its parent. When special edges called suffix links[1] are incorporated into this structure, it is possible to devise construction algorithms that have linear (in the size of the data) time and space complexity, and search algorithms that have linear (in the size of the query) time complexity. Given these remarkable performance properties, it is not surprising that suffix trees have become the defacto standard string index, featuring in tools like MUMmer [5], a global alignment software for genomes developed by Celera Genomics and TIGR (The Institute for Genomics Research). The suffix tree corresponding to the example string aaccacaaca is shown in Figure 2.

From an abstract view-point, the suffix tree can be viewed as an *intra-path*, or "vertical", compaction of the

[1] A suffi x link connects a node representing αS to the node representing S, where α is a single character and S is an arbitrary string.

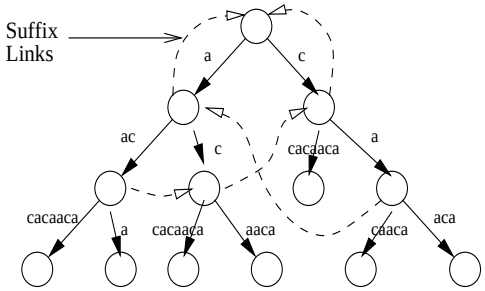

Figure 2. Suffix Tree (for *aaccacaaca*)

original trie since, as mentioned above, unary child nodes are merged into their parents. In this paper, we present a new index structure that is based on a novel *inter-path*, or "horizontal", compaction of the trie. Our motivation stems from the simple observation that there is considerable duplication of patterns *across* the various paths in the trie – for example, in Figure 1, the pattern *cacaaca* appears *thrice* in the trie structure. Eliminating this repetition holds out the promise of significantly reducing the number of nodes in the index and thereby reducing its resource consumption. However, achieving horizontal compaction is a significantly more complex task as compared to vertical compaction. This is because, unlike vertical compaction which is a simple structural merging that is independent of the content of the compacted nodes, horizontal compaction is based on merging character patterns across paths in the trie, thereby immediately running into the risk of generating *false positives* in the compaction process.

1.1. The SPINE Index

In this paper, we present a carefully designed horizontally-compacted trie index structure called SPINE (String Processing INdexing Engine). The SPINE index for the example string aaccacaaca is shown in Figure 3. As seen here, SPINE consists of a backbone formed by a linear chain of nodes representing the underlying data string, with the nodes connected by a rich set of edges for facilitating fast forward and backward traversals over the backbone during index construction and query search. All edges of the index are assigned labels during the construction process, and these labels are used to avoid false positives while traversing the index during the search process.

At a structural level, SPINE provides *all* the standard functionalities provided by suffix trees. Further, it has a variety of other attractive features:

- The entire trie is collapsed into a single *linear* structure, representing the logical extreme of horizontal compaction. Further, the number of nodes is *always* equal to the string length. This is in marked contrast to suffix trees where the number of nodes may go upto *double* the length of the string.

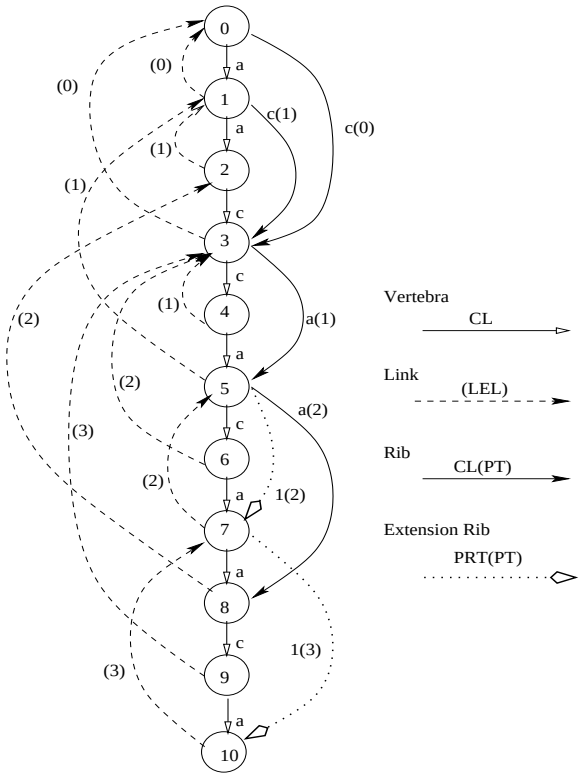

Figure 3. SPINE Index (for *aaccacaaca*)

- Since there is one edge (vertebra) on the backbone corresponding to each character in the string, the data string is *not required* any more once the index is constructed. This property does not hold for most of the other string indexes, including suffix trees.

- A SPINE index can be constructed in an *online* manner, not requiring prior knowledge of the entire data string. Further, a single SPINE index can be used to index multiple different strings, using techniques similar to those employed in Generalized Suffix Trees [1].

- Since index-growth always occurs at the tail of the structure, the node creation order and the node logical order are *identical* in SPINE. The utility of this feature is that it makes SPINE *prefix-partitionable* – that is, given a SPINE index for a string, the index for a prefix of this string is simply the corresponding initial fragment of the index.

  Note that the prefix-partitioning property is *not* supported by suffix trees since a node that is logically high up in the tree may be created much after nodes from lower levels in the tree.

- In comparing Figures 2 and 3, it might appear at first glance that a SPINE index may require more resources than a suffix-tree since it has 11 nodes and 26 edges while the suffix tree has 13 nodes and 16 edges. That is, the node reduction is offset by an increased number of edges. However, as discussed later in this paper, a variety of optimizations can be implemented to minimize the size of the SPINE index such that it is about a third smaller as compared to the equivalent suffix-tree.
- For finding all the matching substrings between two sequences, the number of suffixes processed by SPINE is considerably smaller than those processed by suffix-trees because they process suffixes on an individual basis, whereas SPINE processes them on a *set* basis.
- Finally, due to the simple linearity of SPINE's structure, it is easy to develop efficient buffering policies, a mandatory requirement for good disk performance.

From a performance perspective, we demonstrate through a variety of experiments on real genetic strings, whose lengths are of the order of several millions of characters, that the horizontal compaction approach of SPINE results in significant improvements over vertical compaction. Overall, SPINE takes less space and time to construct, and has better search performance – that is, it wins on both construction and usage metrics. An implication of the lower space requirement is that, for a given memory budget, SPINE is able to process much longer strings than those supported by suffix trees. Even more attractive is that the performance differentials *increase* in moving from fully memory-resident indexes to disk-based implementations.

1.2. Contributions

To summarize, the main contributions of this work are the following:

1. We investigate horizontal compaction of tries and demonstrate that indexes that achieve complete horizontal compaction are feasible.
2. We describe the SPINE index structure and present online algorithms for its construction as well as for searching the index for query strings. We prove that, by virtue of the edge labeling strategy, the searches are guaranteed to not return false positives.
3. We present a variety of optimizations that drastically reduce the memory requirements of the SPINE index.
4. We profile the performance of SPINE against suffix trees over a variety of extremely long genetic strings for both memory-resident and disk-resident scenarios and show that SPINE offers significant benefits with regard to both space and time metrics.

1.3. Organization

The remainder of this paper is organized as follows: The SPINE structure is presented in Section 2, while the construction and search algorithms are described in Section 3 and Section 4, respectively. The specifics of our prototype implementation are outlined in Section 5. Experimental results on the performance of this prototype are highlighted in Section 6. Related work is overviewed in Section 7. Finally, in Section 8, we summarize the conclusions of our study and outline future avenues to explore.

2. The SPINE Index Structure

In this section, we first overview the SPINE index structure and then describe its components in detail.

The central component of SPINE is the "backbone" of nodes connected by forward (or downstream) directed edges called "vertebras", as shown in Figure 3. 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) and "extribs" (dotted lines in Figure 3). 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) are created and used during the SPINE construction process. They provide the ability to process suffixes on a set basis.

2.1. Avoiding False Positives

As mentioned in the Introduction, 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, 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, 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. This is because when we traverse the path from the root for acca we reach Node 5 without violating any rib traversal constraints. Now at 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.

2.2. 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 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, for which S_5 = aacca, s_{52} = ca, $AllSuf_5$ = {aacca, acca, cca, ca, a}, $EndSuf_5$ = { aacca, acca, cca, ca}, $EarlySuf_5$ = { a}.

2.3. 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.

2.4. 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, 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 is an example of this scenario. Finally, as a special case, the root node has no link edge since it is the starting node.

2.4.1. 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:

$$AllSuf_i = EndSuf_i \cup EarlySuf_i \text{ and}$$
$$EarlySuf_i = AllSuf_j(k)$$

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

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 1. Notation

2.5. Ribs

When the SPINE index that has been built for S_i is extended by one more character from the data string, we need to extend all the suffixes of S_i 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.

2.5.1. 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.

2.6. ExtRibs

As mentioned above, we stop the link-chain traversal for rib addition if we find that 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. We therefore take an alternative approach of extending the rib itself through edges called *extribs* (extension ribs). For example, in Figure 3, 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, we take the alternative approach of 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 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 we need to create an extrib, instead of creating it from the destination of the parent rib, 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. All the extribs created for a rib are its children.

2.6.1. 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, an example chain is the extrib from N_5 to N_7, and then from N_7 to N_{10}.

2.7. Proof of Correctness

We have given an informal description of the SPINE index structure above. A formal proof that the valid paths in the SPINE index correspond exactly to the set of substrings that occur in the data string is given in Appendix A.

Another point to note is that it is easy to see from the above discussion that SPINE is prefix-partitionable, i.e. given a SPINE index for a sequence the index for a prefix of the sequence is simply the corresponding initial fragment of the index.

3. SPINE Construction Algorithm

In the previous section, we presented an overview of the SPINE index structure. We now move on to presenting an *online* algorithm for constructing this structure. The pseudo code for the main algorithm is given in Figure 3 (subroutines are described in Appendix B).

We start off with the SPINE index initially consisting of just the root node and then, for each new character in the string, a node is appended to the tail of the index. The vertebra connector to the newly-added node is labeled with the

new character, and the associated links and ribs are created as required.

As mentioned earlier, every node, excepting the root, has a link associated with it. When the first character is appended, a link is created from the new node to the root node. For all subsequent nodes, the following process is followed: The link edge of the immediate predecessor of N_{tail} is traversed upstream. Let the destination node of this 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 = c_{tail}: In this case, a link is created from N_{tail} to the destination node of the vertebra. The LEL of the link is set one greater than the LEL of the last link traversed.

A rib is found with CL = c_{tail}: In this case, if the threshold test does not fail, then a link is created from N_{tail} to the node referenced by the rib, and the LEL of the link is set one greater than the LEL of the last link traversed.

Otherwise, the extrib chain is traversed to find a child extrib with PT $\geq$ LEL of the link. If found, then a link is created from N_{tail} to the destination of that extrib. The PT of the link is set one greater than the PT of the last link traversed. Otherwise, a new extrib is created from the end of the extrib chain to N_{tail} and a link is also created from N_{tail} to the destination node of the last traversed extrib with PRT equal to the PT of the rib which failed the validity test. The link is assigned a LEL which is one more than the PT of the last rib or sibling extrib that has been traversed.

A rib is created from the root node: Here, a link with LEL set to 0 is created from N_{tail} to the root node.

3.1. Construction Example

To help clarify the above discussion, we now describe how the SPINE index is created for the same input string used in Figure 3, i.e. aaccacaaca.

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. An example scenario for each of the conditions in the construction algorithm given in Figure 4 is discussed below.

```
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. edgeFound = FALSE
06. WHILE (NOT edgeFound)
07.     IF (N_curr ≠ NULL )
08.         l = Most recently traversed link
09.         Check for a c_tail vertebra/rib at N_curr
10.         IF (a matching edge e is found)
11.             IF (e is vertebra)
12.                 AddLink(N_tail, e.Dest, l.LEL + 1)
13.             ELSE IF (e is a rib)
14.                 IF (l.LEL > e.PT)
15.                     HandleExtribs(l.LEL, e.PT)
16.                 ELSE
17.                     AddLink(N_tail, e.Dest, l.PT + 1)
18.             edgeFound = TRUE
19.         ELSE
20.             AddRib(N_curr, N_tail, c_tail, l.LEL)
21.             N_curr = Link(N_curr).Dest
22.         END-IF
23.     ELSE // link chain ends
24.         AddLink (N_tail,N_root,0)
25.         edgeFound = TRUE
26.     END-IF
27. END-WHILE
```

Figure 4. SPINE Construction Algorithm

CASE 1: Vertebra Exists (Line 11)
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 LEL of 1 (= LEL of last traversed link + 1).

CASE 2: Rib With Required PT Exists (Line 16)
This case occurs when there already exists a rib/vertebra for c_{tail} with sufficient PT. Consider appending N_4. In this case, we find that a rib for c with sufficient PT exists at N_0. Hence a link is created from N_4 to N_3 (the destination of the rib) with a LEL of 1 (= LEL of last link traversed + 1).

CASE 3: Rib Creation (Line 19)
This case occurs when there exists no rib/vertebra for c_{tail}. Consider appending N_3. Traverse the link of N_2 to reach N_1. Since there exists no rib/vertebra for c, create a rib from N_1 to N_3 and assign it a LEL of 1. Now traverse the link of N_1 to reach N_0. Again, since no rib or vertebra exists for c, a rib is created for character c from N_0 to N_3 with PT equal to 0. Since the

root node has no link, we end the process by creating a link from N_3 to N_0 with LEL = 0.

CASE 4: ExtRib Creation (Line 15)

This case occurs when there exists a rib whose PT is less than the desired value. Consider appending N_7. Traverse the link of N_6 to reach N_3. At N_3, there exists a rib for character a but with PT of 1 which is less than the LEL of the last traversed link($= 2$). 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 PT and PRT are set to 2 (LEL of the last traversed link) and 1 (PT of the parent rib), respectively. Then, a link is created from N_7 to N_5 (the last traversed rib/extrib with the same PRT as the newly created extrib) with a LEL of 2 (= PT of last traversed rib/extrib with same PRT + 1).

4. Searching with SPINE

In this section, we discuss how a SPINE index can be used for efficient searching. For illustrative purposes, we will assume a complex matching operation wherein the goal is to find, given a data string S1 on which a SPINE index has been built, and a query string S2, all maximal matching substrings, including repetitions, between S1 and S2, whose lengths are above a threshold value. A practical application of this matching operation is in establishing local alignments across genetic strings.

For example, given the following strings S1 and S2, and a threshold value of 6

S1 *acacc***gacgat***acga***gattacgaga***cgagaatacaacag**
S2 *catagagagac***gattacgaga***aaacgggaaa***gacgat***cc**

the output should contain the substrings shown in boldface.

For the above operation, the SPINE matching would proceed as follows: To start off, the entire query string is searched for in the SPINE index of the data string. 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 in the matched part of the query string, and the process is repeated again. The shorter suffixes are reached by traversing the link chain upwards.

The procedure for finding a match is as follows: We start from the root node and traverse the forward edges (vertebras, ribs and extribs) according to the characters in the query string. 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 $\leq$ PT of the rib. If this test fails, then this rib's extrib chain is followed until either the extrib chain ends, or we find the child extrib whose PT is greater than or equal to the current pathlength and having a PRT which is equal to PT of the rib which failed the test.

The intuition behind our searching scheme is simple: Each valid path starting from the root to a node corresponds to some suffix of the string on the backbone till that node. And while more than one suffix might terminate at a node, each such suffix would be of a different length. So, at a given node if it is valid to traverse a rib after a pattern p (suffix till that node) of length k, then it has to be valid after a pattern q whose length is less than k and which ends at the same node, because q would be a suffix of p.

The above matching process finds the first occurrence of a match in the data string. But our goal is to find *all* occurrences of the match. This is achieved using a simple technique that exploits the link property that a link with *LEL v* from node N_a to node N_b indicates that a string of length *v* above N_a is the same as the string of length *v* above N_b. Specifically, after we find the first occurrence of the match, 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 and have an LEL greater than length of pattern being searched. 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. Searching in the target node buffer is performed in binary fashion to improve the performance.

To clarify the above, consider Figure 3 with a query string 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 LEL = 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 LEL 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 string. 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.

While we could, in principle, search for all occurrences of a matching pattern immediately after it is found, this would be wasteful since it would require a traversal of the backbone for each matching pattern. Instead, we defer this step until the first occurrences of all matches are found, and then, in one single final sequential scan of the backbone, the repeated occurrences of all matching patterns are concurrently found.

The detailed pseudocode to find the first occurrence of the query sequence in the data sequence is described in Appendix C.

4.1. Comparison with Suffix-Trees

Similar to SPINE's use of links, suffix-trees use *suffix links* to assist in finding the suffixes of the matched substrings. But, the number of suffixes checked by suffix-tree search algorithms is *far more* than those 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 a complete match is found or there are no more suffixes remaining to be checked.

In SPINE, however, each node N_i in a link chain represents a *set* of suffixes, namely $EndSuf_i$. Therefore, only *one check* is sufficient for all the suffixes in that set, reducing the computational effort.

To make the above analysis clear, consider the index structures shown in Figures 2 and 3. Here, assuming that while matching accaa a mismatch is found in the suffix-tree after matching acca, then the next suffix to be checked will be cca (length 3) i.e. the one indexed by the destination node of the suffix link. On the other hand, in SPINE, the link from Node 5 *directly* points to Node 1 which represents the suffixes of length 1 or less. This means that the (unnecessary) checks for suffixes of length 3 and length 2 are not made. Therefore, for large strings, a very small number of suffixes are actually checked in the SPINE index.

5. Implementation Details

We have developed a prototype version of SPINE, and in this section, we discuss its implementation details. While SPINE is general in its applicability, for ease of presentation, we will assume in the following discussion that it is DNA genomic strings, which are over an alphabet of size four, that are being indexed; proteomes, which are over an alphabet of size twenty, are discussed at the end of the section.

Our implementation strategies are based on our experience with a variety of DNA genomes, each of which is several million characters in length. In particular, we will present results for the following representative genomes:

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

CEL : C.Elegans bacterial 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.

Field Name	Space (Bytes)	Count	Total (Bytes)
CharacterLabel	0.25	1	0.25
VertebraDest	4	1	4
Link Dest	4	1	4
Link LEL	4	1	4
Rib Dest	4	3	12
Rib PT	4	3	12
ExtRib Dest	4	1	4
ExtRib PT	4	1	4
ExtRib PRT	4	1	4

Table 2. Index Node Content

The information associated with each node of the SPINE index and the associated space requirements are shown in Table 2, corresponding to storing one vertebra, one link, a maximum of three ribs (for DNA alphabet), and one extrib at the node. As can be seen from the table, with a straightforward implementation, the worst-case space required by each node is huge (48.25 bytes). However, the SPINE index exhibits a variety of both structural and empirically-observed features using which the actual space required can be drastically reduced – in fact, as we will show next, it can be brought down to *less than 12 bytes* when all these features are taken into account.

5.1. Node Size Optimizations

In this section, we present the optimizations which reduce the space requirements of SPINE index.

5.1.1. Implicit Vertebra Edge

Since, as mentioned earlier, SPINE grows sequentially at the tail of the backbone, the physical order and the logical order of the nodes are identical. We can take advantage of this feature to not explicitly represent the vertebra edge destination since the neighboring nodes are physically contiguous, i.e. the *Vertebra Dest* field can be eliminated.

5.1.2. Small Numeric Label Values

Table 3 gives the maximum value observed for the various numeric labels (PT, LEL, PRT) when the SPINE index was constructed on the representative biological genomes mentioned above. As can be observed here, the label values *never exceed* 25000 even for very large genomes like the human chromosomes. Therefore, only two bytes, rather than four, need to be allocated for the length fields.

However, to ensure that the index works robustly, we have a mechanism in place to handle even those rare cases where the numeric values may exceed 65536 (the maximum value that can be represented in two bytes). We allocate separate entries for these cases in an *overflow table*. The node space normally used for storing the label value is now used

Genome	Max Value
ECO	1785
CEL	8187
HC21	21844
HC19	12371

Table 3. Maximum Label Values

to index into the overflow table, and a one bit flag is used in the main node structure to indicate whether the space is storing a value or a pointer.

5.1.3. Sparse Rib Distribution

While all nodes have upstream edges (links), the same is not true with respect to downstream edges (ribs and extribs). In fact, we have found that only around *30 to 35 percent* of the nodes actually have any downstream edges emanating from them – Table 4 shows the distribution of their number for the various genomes. Specifically, the columns labeled 1 through 4 represent the percentage of nodes having that many forward edges emanating from them, with the maximum corresponding to having the full complement of downstream edges (3 ribs and 1 extrib).

Genome	Number of Ribs				Total
	1	2	3	4	
ECO	15%	9%	6%	4%	33%
CEL	15%	8%	6%	4%	33%
HC21	14%	8%	6%	4%	32%
HC19	13%	7%	5%	3%	28%

Table 4. Rib Distribution across Nodes

The reason for this distributional behavior is that after some length of the data string has been processed, the remaining part mostly contains *repetitions* of previously occurred patterns, and therefore fresh downstream edges are rarely created. Based on this observation, we do not allocate space for downstream edges at every node, since considerable space would be wasted. Instead, we store information about the links and the downstream edges separately in a *Link Table* (LT) and a *Rib/Extrib Table* (RT), respectively. One entry for each character in the string is allocated space statically in the *LT*, while space for downstream edges is allocated dynamically in the *RT* for only those nodes from which a rib/extrib emanates. Therefore, the total number of entries in the RT is less than 35 percent of that in the LT.

Further, from Table 4 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 downstream edges. Therefore, to avoid the space wasted for the edges which are not present, we use *multiple* RTs. Specifically, there is one RT for each possible fanout, resulting in four RTs in total: *RT1, RT2, RT3, and RT4*.

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

5.1.4. Final Node Layout

Based on the above discussion, the optimized implementation of the SPINE index consists of a **Link Table** (LT) and four **RibTables** (RTs), whose entries are shown in Figure 5. The LT contains one entry for each node (character) in the string. It stores its LEL 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 RTs (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 RT entries only.

Each node features in at most one RT table. A RT entry for a node stores the destination node of the link from that node and also the destination nodes (the *RD* fields) and the threshold values (the *PT* fields) of all the ribs/extrib emanating from the node. And, lastly, the *PRT* field denotes the PRT value of the extrib.

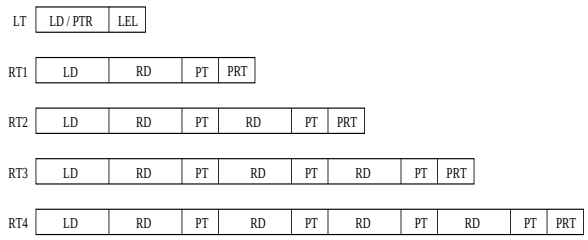

Figure 5. Optimized SPINE Implementation

By implementing all the above optimizations, the net effect is that the average node size in SPINE is *less than 12 bytes*, that is, the index takes upto 12 bytes per indexed character. The advantage of smaller node sizes is reflected not only in space occupancy but also in improved construction and searching times, as is quantitatively demonstrated in the following section.

5.2. SPINE Implementation for Proteins

The above implementation focused on DNA strings which have an alphabet size of 4. When we consider proteins strings, where the alphabet size increases to 20, we observed that the numeric label values are *even smaller* than those found with DNA sequences. Our experiments were conducted with the E.Coli Residue (1.5 M), Yeast Residue (3.1 M) and Drosophila Residue (7.5 M) proteomes. Moving on to the rib distribution, we observed that here too there

is a steep decay in the percentages of nodes having multiple ribs. And again, the total number of nodes with any rib/extrib is less than 30%. Therefore, the overall behavioral characteristics of proteomes are similar to that of genomes with the only practical difference being that each character label requires 5 bits to code as opposed to the 2 bits used for DNA.

6. Experimental Analysis

We conducted a detailed evaluation of the performance of the SPINE index prototype, and these results are presented in this section. Our experiments were conducted with the same set of genomes mentioned earlier in this paper (i.e. E.Coli, C.Elegans, HumanChromosome 21, and HumanChromosome 19). For comparison purposes, we also evaluated the performance of the suffix tree, hereafter referred to as ST – the code base was taken from the MUMmer software to reflect an industrial-strength implementation.

Our experiments were conducted on a Pentium IV 2.4 GHz machine with 1 GB RAM, 40GB IDE disk and running Linux 7.3 operating system. The performance metrics in our experiments were the following:

Index Construction Time: 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 complex matching operation discussed in Section 4, wherein we need to output all maximal matching substrings, including repetitions, between the source strings.

In the following discussion, we first consider an environment wherein both the data string and the index structure are completely memory-resident, and then move on to presenting results for disk-resident indexes.

6.1. In-Memory Environment

The performance of ST and SPINE with regard to index construction times are shown in Figure 6. Firstly, note that the indexes take less than two seconds construction time per Mbp, which means that with sufficient resources, a complete in-memory index construction of the human genome (approximately 3Gbp in length) can be done in under two hours. Second, SPINE takes only marginally lesser time to construct than ST, especially for longer strings. This is not surprising since all operations are done in memory and therefore the structural differences do not really play a role in determining the construction time. But, these features do show up with regard to the *maximum string length* that can be successfully handled for a given budget. This is evident in Figure 6, where no results are shown for ST with regard to the HC19 string as it ran out of memory due to its larger space requirements. In contrast, SPINE was able to complete the index build successfully – in general, SPINE can handle approximately *30 percent more string length* than the maximum that can be supported by ST.

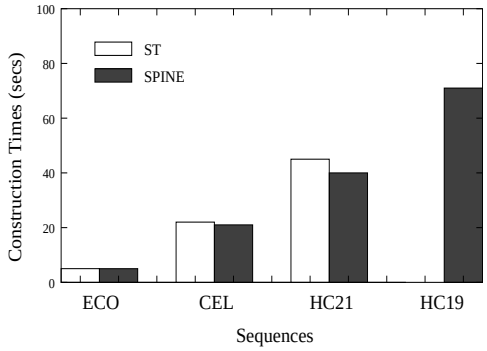

Figure 6. Index Construction Times (In Memory)

Moving on to the search times, Table 5 gives the times required to find all the exactly matching substrings (including all multiple occurrences) for SPINE and ST for various genome pairs. We observe here that SPINE takes around 30 percent lesser time than ST. This is entirely due to its efficiency in handling a much smaller number of suffixes, as described earlier in Section 4.1, and quantitatively shown in Table 6.

Data Seq	Query Seq	ST	SPINE
ECO	CEL	20	16
CEL	HC21	45	31
HC21	CEL	26	17
HC21	HC19	83	54
HC19	HC21	–	30

Table 5. Substring Matching Times (secs)

Data Seq	Query Seq	ST	SPINE
CEL	ECO	3515	2119
HC21	ECO	3514	2163
HC21	CEL	15077	8701

Table 6. Number of Nodes Checked (In 1000s)

We hasten to add here that while the results we show here are for complete genomes in order to demonstrate scalability, the same performance differences held even when the query strings were much smaller (for example, of length 1K).

6.2. Performance on Disk

We now move on to assessing the performance of SPINE and ST on disk. Note that while SPINE can be expected to have a basic advantage due to its smaller node size, the more important issue here is the *locality* of the accesses made by the index structures.

To study their behavior, we constructed SPINE and ST indexes on disk without doing any extra disk specific optimization. The graph in Figure 7 shows the time taken to construct the indexes for the various genomes on disk. We see here that SPINE takes almost half the time as required by ST to construct the index on disk. Note that this cannot be attributed solely only to the smaller-sized nodes since that would have at best reduced the time by a factor of about 30%. The additional 20% improvement arises due to the better locality exhibited by SPINE.

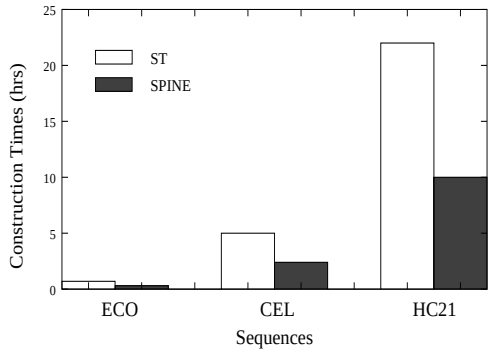

Figure 7. Index Construction Times (On Disk)

We investigated the issue of locality further and an interesting feature that we observed in the SPINE index 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 8, which shows the distribution of the link destinations for different data strings. This indicates that while constructing the SPINE, the upstream nodes would be accessed more than the downstream ones.

Hence this suggests a *simple buffering strategy* for SPINE when sufficient memory is not available: *"Retain as much as possible of the top part of the Link Table in memory"*.

Moving on to the index search times, we observed that the time required to obtain all the exactly matching substrings also improved by a factor of two with SPINE as compared to ST. This is explicitly shown in the speedup numbers of Table 7 for the various genome combinations.

Due to space limitations, we do not present performance results for protein strings here, but our experiments with these strings showed that the SPINE construction times for proteins also scaled linearly with the string lengths, and that the search times are independent of the data string length. Overall, SPINE works as well with protein strings as it does with DNA strings.

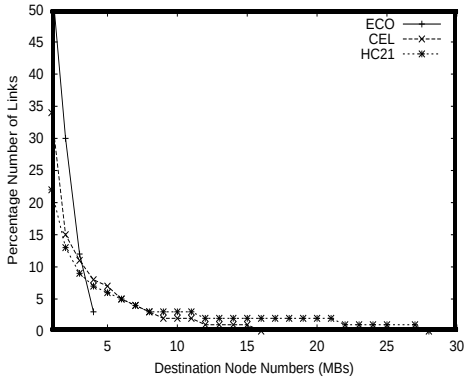

Figure 8. Link Distribution over the Backbone

Data Seq	Query Seq	Speedup
ECO	CEL	52.1%
ECO	HC21	49.8%
CEL	HC21	49.2%
HC21	CEL	52.8%
HC21	HC19	51.1%

Table 7. Substring Matching (On Disk)

7. Related Work

A rich body of literature exists with regard to vertically compacted trie indexes such as suffix trees. The primary focus of this research has been on optimizing the space occupied by the tree nodes – however, these optimizations typically adversely impact either the performance or the functionality. For example, Kurtz [2] proposed an implementation that requires 12.5 bytes per indexed character for DNA sequences. However, suffix trees built using this technique take more time for construction and searching times are also impacted due to the comparatively larger number of edge traversals.

An even more space-efficient implementation, called Lazy Suffix Trees [3], has been recently proposed, taking only 8.5 bytes per indexed character. 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.

In contrast to vertical compaction, there is almost no prior work available with regard to horizontal trie compaction. The only exception that we are aware of in this regard is *DAWGS* [7], which require around 34 bytes per

character for DNA sequences [2]. A compacted version of DAWGS, called *CDAWGs* [8] was proposed, but that too requires more than 22 bytes per indexed character [2].

More importantly, they are unable to achieve complete horizontal compaction due to their technique for eliminating false positives. Further, they lack position information of the matching pattern in the data sequence because their nodes do not correspond to the character positions in the sequence.

Recently, in order to make suffix-tree construction on disk efficient, a partition-based technique was proposed in [10]. However, this algorithm is predicated on dispensing completely with the suffix links that are essential for retaining the linear time construction complexity – as a result, the algorithm in [10] has quadratic complexity. Further, the removal of the links makes approximate matching and substring 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.

An elegant two-level search technique called MAP was recently proposed in [12], wherein a preprocessing phase using a very small approximate index is used to first filter out those regions of the data string that potentially contain matching entries, and then a seed-based approach is used on the filtered regions. MAP was reported to be between one to two orders of magnitude faster than BLAST, the popular genomic alignment tool [12]. Both MAP and BLAST give approximate answers, whereas SPINE and ST provide exact answers. Further, the performance improvement through indexes is usually substantially more albeit at the cost of increased resource consumption.

8. Conclusions

In this paper, we have proposed the SPINE index data structure, which achieves a complete horizontal compaction of the basic trie structure used for indexing long strings, and ensures that the number of nodes in the index is equal to the number of characters in the underlying data string. To the best of our knowledge, this is the first string index with these properties, and is in marked contrast to suffix-trees, the defacto standard string indexing structure. A rich set of forward and backward edges are employed in SPINE to ensure that all suffixes of the data string are captured in the index structure. Further, the false positives that inevitably resulted from the trie compaction were eliminated through a simple but powerful numeric labeling strategy that constrains when the index edges can be traversed. Finally, the SPINE index is prefix-partitionable, a property not shared by suffix-trees.

We provided detailed algorithms for both online construction of the SPINE index as well as for performing complex searching operations on the resulting indexes. A feature of the search algorithm is that it considerably reduces the number of suffixes that have to be examined during the alignment process. 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 12 bytes per indexed character, comparing favorably with the 17 bytes taken by standard suffix tree implementations.

A performance evaluation of SPINE against ST (Suffix-Tree) over a variety of very long genetic strings, including human chromosomes, showed that significant speedups were obtained for the searching operations, for both memory-resident and disk-resident scenarios. It was also observed that along with 30 percent lesser index size, SPINE exhibits much higher node locality than ST, resulting in a more efficient disk-based implementation. Finally, it was shown that a very simple buffering strategy was sufficient for SPINE to be able to take advantage of the locality observed in our experiments. In summary, SPINE appears to be a viable alternative to suffix-trees for string indexing.

References

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

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

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

[4] M. Crochmore and R. Verin, "Direct Construction of Compact Acyclic Word Graphs", *Proc. of Symp. on Combinatorial Pattern Matching*, 1997.

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

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

[7] A. Blumer et al., "The Smallest Automaton Recognizinng the Subwords of a Text", *Theoretical Computer Science*, 40:31-55, 1985.

[8] S. Inenaga et al., "On-Line Construction of Compact Directed Acyclic Word Graphs", *Proc. of Symp. on Combinatorial Pattern Matching*, 2001.

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

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

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

[12] T. Kahveci and A. Singh, "MAP: Searching Large Genome Databases", *Pacific Symp. on Biocomputing*, 2003.

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

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

APPENDIX

A. Proof of SPINE Correctness

We present here the formal proof demonstrating that the SPINE index does not result in any false positives. The notation defined in Table 1 is used in the proof:

Lemma 1: *An incoming rib or extrib into node N_i is added only when N_i is appended to the backbone and at no other time.*

Proof: When appending 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 Section 2, are extended by using ribs/extension ribs during the construction process. And according to the construction algorithm, we create a rib from node N_j ($j < i$) to N_i if we have to extend some suffix s_{jk} (which is identical to $s_{(i-1)k}$) to obtain $s_{i(k+1)}$. In other words, ribs are created to accomodate the suffixes that are newly created due to the addition of the new character at the end of the string. For example, $s_{(i-1)k}$ is a suffix of both S_j and S_{i-1} but $s_{i(k+1)}$ occurs only in S_i. So, the incoming ribs of node N_i are created only when that node is being appended to the vertebra and at no other time; each one of these ribs correspond to different suffixes of S_i.

The above proof holds true for extribs as well because the destination of an extrib is always the node that is being appended.

Theorem 1: *The addition of ribs and extension ribs do not create false positives in SPINE.*

Proof: We use an inductive technique for this proof. Assume that there are no *false positives* in SPINE until S_i and that we are now extending this index with a character c. Appending a character to the SPINE index involves 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 node N_{i+1} is appended to the backbone. Note firstly that all the suffixes of S_i in $EndSuf_i$ are *automatically* extended by the newly added vertebra. By Lemma 1, we know that all the paths to any node N_i are created when N_i is appended to the backbone and they represent suffixes of S_i and therefore we are extending only the valid suffixes and none other. Hence the addition of the vertebra does not create any false positives.

Next we consider the additional paths created by addition of ribs and extribs. For extending the other suffixes of S_i which belong to $EarlySuf_i$, we traverse the link chain starting from N_i as explained in Section 2. Suppose from N_i, we traverse the link and reach N_j (i.e. $j = Link(N_i).Dest$), and $Link(N_i).LEL$ is l. By definition of a link we have $AllSuf_i(l) \equiv AllSuf_j(l)$. Now at N_j,

Case 1: No Rib or Vertebra exists for c

Since there exists no rib or vertebra for c at N_j, none of the suffixes in $EndSuf_j$ have been extended by c. Further, note that only the suffixes in $EndSuf_j(l)$ and not all the suffixes in $EndSuf_j$, need to be extended by c.

According to the construction algorithm, a new rib is created at N_j for c and given a PT of l. This PT indicates that the rib is valid only for suffixes in $EndSuf_j(l)$. Thus the PT of the rib ensures that only the required suffixes in $EndSuf_j$ are extended and that any valid path ending in a rib at N_{i+1} represents some suffix of S_{i+1}.

From Lemma 1, we have that ribs coming into a node are created only when that node is being appended to the vertebra and at no other time. And from the above discussion we have seen that any valid path ending in a rib at a node represents some suffix of the string on the backbone above that node. Therefore, ribs do not create any false postives.

Case 2: Extension Rib creation

Let $dest = Rib(N_j).Dest(c)$ and $pt = Rib(N_j).PT(c)$. This case occurs when $pt < l$, which indicates that this rib is valid for $EndSuf_j(pt)$ but not s_{jl}. Hence we create an extension rib from N_{dest} to N_{i+1} and it is given a PT of l. This PT ensures that only the required suffixes are extended as explained in Case 1.

Now there can be multiple ribs ending at N_{dest} and we need to identify to which one of them the newly created extension rib corresponds. We add additional information PRT to the extension which is nothing but the PT of the rib which failed the test. This works because all the ribs ending at a node are guaranteed to have different PT values 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 or extribs to obtain the SPINE index for S_{i+1} does not create any false positives.

B. Subroutines for SPINE construction algorithm

The details of the subroutines mentioned in Figure 3 are given below in Figure 9.

AddLink($Source, Dest, Label$**)**
01. Create a link l from $Source$ to $Dest$
02. $l.LEL = Label$

AddRib($Source, Dest, CharacterLabel, PathLengthThreshold$**)**
01. Create a rib r from $Source$ to $Dest$
02. $r.CL = CharacterLabel$
03. $r.PT = PathLengthThreshold$

HandleExtribs(l_{LEL}, r_{PT}**)**
01. Search for an extrib e such that
02. ($e.PT >= l.LEL$) & ($e.PRT = r.PT$)
03. IF (there exists such an e)
04. AddLink($N_{tail}, e.Dest, s.LEL + 1$)
05. ELSE
06. Create extrib e from extrib chain end to N_{tail}
07. $e.PT$ = $l.LEL$
08. $e.PRT$ = $r.PT$
09. Let $lrib$ = last traversed rib or extrib
10. AddLink($N_{tail}, lrib.Dest, lrib.PT + 1$)

Figure 9. Subroutines

C. Exact Match Algorithms

Figure 10 gives the pseudocode to find the first occurrence of the longest prefix of a given query sequence in the data sequence. To find all the substring matches, this algorithm is used as explained earlier in Section 4.

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.PT $>= travLen$
11. $travLen++$;
12. $curr$ = R.Dest;
13. ELSE
14. PRT = R.PT;
15. $temp$ = R.Dest;
16. extribFound = FALSE;
17. WHILE ((NOT extribFound) and (extrib exists at N_{temp}))
18. IF extrib.PRT = PRT
19. IF extrib.PT $>= travLen$
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 10. Longest Prefix Match