MSc Computer Science project report, Department of Computer Science and Information Systems, Birkbeck College, University of London

De novo sequence assembly using cloud infrastructure

Supervisor: Dr. Nigel Martin

Author: Adam Serafini

2012

Contents

1. Introduction

De novo sequence assembly is a process which attempts to reconstruct a complete DNA sequence from many short fragments. The process is necessary as modern DNA sequencing machines cannot determine whole DNA sequences in one read. To work around this problem, DNA sequences are duplicated and then fragmented in a process called “shotgun sequencing”.

The output from the “shotgun sequencing” process is a large number of fragments (strings) which must be assembled, using computers, into the original sequence (superstring). In this project, I implement a well-known abstraction of the assembly problem and test my implementation on high performance cloud computers available via Amazon’s EC2 network.

Historically, the assembly process was exclusively performed using ‘Overlap graphs’ which I use in this implementation and are described later in this report. However, as output from sequencing technology has shifted from low volume/long fragment output to high volume/short fragments, it has become necessary to develop new methods which reduce the main memory requirements of assembly software. For this reason, ‘de Bruijn’ graphs, which use significantly less memory than overlap graphs have recently gained in popularity among the available tools. However, to achieve their superior memory footprint, de Bruijn graphs sacrifice some information about the composition of each read, and this information must be ‘reintroduced’ in a downstream process.

My results show that the classic Overlap model of assembly *is* still a viable method for assembling very small genomes, particularly using the high performance computing power available on Amazon’s EC2 network. Furthermore, the suffix tree approach is confirmed as an efficient method for computing pairwise overlaps between large sets of strings, and is shown to be significantly more effective than the naïve method using C++ inbuilt string functions. However, more work is needed to reduce the memory requirements of the suffix tree approach.

1.1 Basic String Definitions

Fundamentally, the DNA sequence assembly problem can be framed as a string problem in computer science. I will now define some basic definitions and nomenclature that are used throughout the project report to describe strings and algorithms on strings. Gusfield defines a string and substring as follows:

“A string `S` is an ordered list of characters written contiguously from left to right. For any string `S`, `S[i..j]` is the (contiguous) substring of `S` that starts at position `i` and ends at position `j` of `S`.” [1]

The number of characters in a string will normally be denoted by `n`, however sometimes it will be necessary to disambiguate which string is being referred to, in which case `|S|` can also refer to the length of string `S`.

Note that C and C++ use ‘zero-based’ string indexing, meaning that access to the first character of a string is indexed by the element zero. However, string algorithms in Gusfield [1], Ukkonen’s paper [2] and a further paper by Turner [3] use ‘one-based’ indexing, meaning that a string of `n` characters is represented as `S[1..n]`. As algorithms from these sources provide the foundation of the submitted implementation I have followed their lead and also use 'one-based' indexing in this report.

In addition to the basic string definitions:

"`S[1..i]` is the *prefix* of string `S` that ends at position `i`, and `S[i..n]` is the *suffix* of string `S` that begins at position `i`" [1]

"`S[i..j]` is the empty string if `i > j`" [1]

"For any string `S`, `S(i)` denotes the `i`th character of `S`" [1]

Two characters are said to *match* if they are the same. A string S_i matches another string S_j if the characters at each position match, eg. $S_i(1) = S_j(1) \ldots S_i(n) = S_j(n)$.

Finally, the concept of overlapping strings is fundamental to this project. Gusfield defines the intuitive concept of an overlap as the *suffix-prefix match* between two distinct strings:

"any suffix of S_i that matches a prefix of S_j is called a suffix-prefix match of S_i, S_j" [1]

For example, the suffix-prefix match of "urgent" and "gentle" is the string "gent".

1.2 The Shortest Common Superstring Problem

After downloading and studying the available open source implementations of de novo sequence assemblers from all four categories described in the project proposal (greedy, overlap graph, de Bruijn graph and hybrid), it was clear that a writing a fully-fledged assembler was unrealistic within the time constraint.

For this reason, I chose to focus on a simplified model, called the shortest common superstring problem which is commonly referred to as an approximation of the assembly process [4] [1]. I will refer to it as the *SCS* problem during the report.

A superstring is defined as follows by Gusfield: "given a set of k strings $P = (S_1, S_2 \ldots, S_k)$, a *superstring* of the set P is a single string that contains every string in P as a substring" [1]. A naïve superstring would simply be a concatenation of the set of strings in P.

The shortest superstring of P is the single string that contains every string in P as a substring *and* has minimum length. This optimization is known to be NP-hard [5], and can be formulated as a travelling salesman problem using overlap graphs which I describe later in the report. Consequently, heuristic methods are almost always used to solve this problem, especially considering the volume of data in biological applications where a set may contain upwards of 100 million strings.

1.3 Structure of the Report

Constructing the overlap graph for a set of strings requires that the overlap between each distinct pair of strings is known. This problem is referred to as the "all-pairs suffix-prefix problem" [1]. The general suffix tree data structure allows this problem to be solved efficiently in linear time; Section 2 of the report is dedicated to describing suffix trees;

Section 3 describes an algorithm for their efficient construction and representation. Section 4 expands on this to describe *general* suffix trees for a set of strings and their application to overlap detection.

Section 5 introduces the overlap graph and describes an efficient implementation of the greedy SCS algorithm.

Section 6 demonstrates the correct functioning of the implementation and benchmarks against the naïve method. Amazon's cloud computing infrastructure is used to test the program on a data set of simulated fragments extracted from the Swinepox virus.

Section 7 contains an evaluation of the project, reflecting on aspects of the implementation that could be improved.

Appendix A contains information about running and compiling the program in Windows. Appendix B contains a copy of the program code. Finally, Appendix C contains a bibliography.

1.4 Use of C++

This project was implemented in C++, a design decision that was taken early on to enable rapid progress and experimentation. Factors that contributed to this decision included my familiarity with the programming language, existing implementations of suffix trees in C/C++ and the excellent Microsoft Visual Studio IDE for C++. The comprehensive debugging and profiling features of Visual Studio were relied on extensively during the implementation.

2. Suffix Trees

2.1 Description

A suffix tree of string S, represents all suffixes of S and allows efficient solutions to many complex and important string problems. The data structure can be built in $O(n)$ time where n is the number of characters in string S. The formal definition (duplicated from [4]) follows:

Definition 2.1: Let $S = S(1) \ldots S(n) \in \sum^n$ be a string, where $S(n)$ is a unique terminal symbol that does not appear elsewhere in the string (denoted by '\$'). A directed tree $T_t = (V, E)$ with a root r is called a suffix tree for S if it satisfies the following conditions:

1. The tree has exactly n leaves which are labelled $1, \ldots, n$.
2. The edges of the tree are labelled with symbols from $\sum$
3. No two edges leaving a node may start with the same symbol.
4. The path label from the root to the leaf i is labelled $S(i) \ldots S(n)$, where the 'path label' of a leaf is the concatenation of edge labels starting at the root and ending at the leaf.

For example, a suffix tree for the string *xabxa\$* is shown in Figure 2.1:

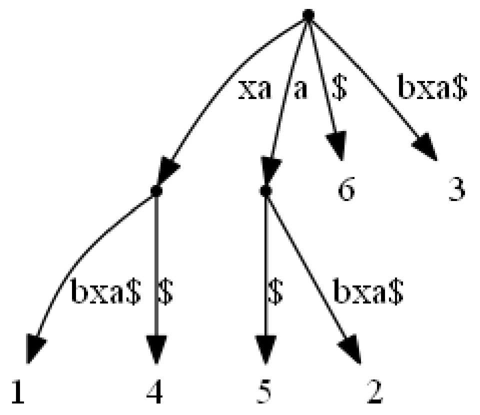


Figure 2.1: *The suffix tree for string xabxa$.*

Let us check that it satisfies the conditions in Definition 2.1:

1. The tree has 6 leaves, and the string *xabxa$* has 6 characters.
2. The edges of the tree are labelled only with symbols in the string *xabxa$*.
3. The first character of each outgoing edge from any node is distinct. In the example of the root, the four outgoing edges start with letters x, a, c and b.
4. In the example of `i = 2`, following edge labels from the root to leaf `2` yields a path label of *abxa$*. This matches the substring of *xabxa$* starting at the second character, eg. string `S(2), S(3), S(4), S(5), S(6).`

2.2 Exact String Matching using Suffix Trees

A common problem is immediately solvable using suffix trees: exact string matching. Given a pattern `P` of length `n` and a string `S` of length `n`, the exact match problem finds all occurrences of `P` in `S`. Having built a suffix tree for `S`, one can match the characters of `P` along a unique path in `S` until either `P` is exhausted or no more matches are possible. If `P` is exhausted, every leaf in the sub-tree below the point of the last match is numbered with a starting location of `P` in `S`. If `P` Is not exhausted and no more matches are possible, `P` does not occur anywhere in `S`.

For example, using the suffix tree in Figure 2.1, to find all occurrences of substring `P` = "xa" in string `S` = "xabxa$" we follow the path "xa", leading us to the node labelled `v` in the following diagram:

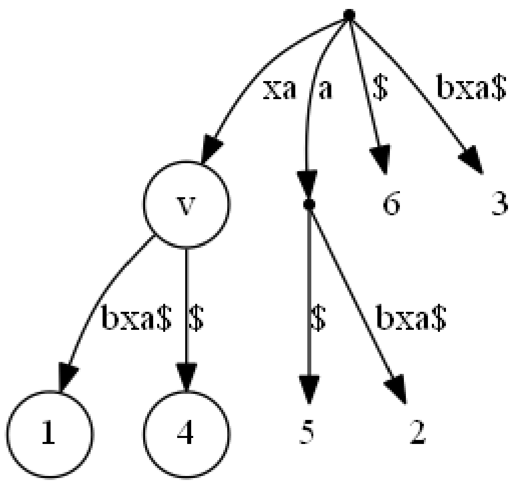


Figure 2.2: *Exact matching of* `P` *= "xa" in* `S` *= "xabxa$". As* `P` *is exhausted, the leaf labels 1 and 4 correspond to the starting locations of xa in string xabxa$.*

Once the suffix tree is built in `O(|S|)` time, patterns can be checked against the tree in `O(|P| + k)` time, where `k` is the number of occurrences of `P` in `S`. This neat property is useful whenever a large number of short strings must be matched against a much longer (but static) string, a common requirement in bioinformatics applications. Indeed, I used this technique in the testing phase of the project to verify that all strings in the set provided to the software were represented in the resulting superstring.

2.3 Existing Implementations

Efficient suffix trees appear to be notoriously hard to implement, as evidenced by the surprisingly few existing implementations available in the public domain. According to Gusfield:

"suffix trees have not made it into the mainstream of computer science education, and they have generally received less attention and use than they might have expected. This is probably because the two original papers of the 1970s have a reputation for being extremely difficult to understand". [1]

This sentiment is echoed consistently in the published literature concerning suffix trees. For example:

"In spite of their basic role for string processing, elementary books on algorithms and data structures barely mention suffix trees, and never give efficient algorithms for their construction. The reason for this is historical: starting with the seminal paper by Weiner, suffix tree construction has built up a reputation of being overly complicated". [10]

The textbook "Algorithmic Aspects of Bioinformatics", dedicates a whole chapter to suffix trees but declines to describe an efficient algorithm for their construction "since the known algorithms for linear-time suffix tree construction are technically quite involved". [4]

For this reason, before commencing the considerable effort involved in coding my own suffix tree, I investigated the existing implementations available on the internet in the hope that one might be re-purposed for use in my application. Unfortunately no existing implementation was an exact fit for my use case, but I briefly summarise the existing suffix trees (that I am aware of) here:

Mark Nelson [6]: originally published in Dr. Dobbs journal, Mark Nelson has a copy of this article on his blog, along with the source code in C++. However, after compiling and testing the program I found that it violated Property 4 in Definition 2.1: specifically the leaf labels were not labelled in such a way that the path label from root to leaf `i` corresponded with the substring `S[i..n]`.

Dotan Tsadok[7]: this implementation in ANSI C was written as a Haifa university student project. I studied this implementation closely, and borrow one key idea from it: the system for representing nodes and edges which I describe in Section 3.5. However, the implementation was for a single input string only and not a generalized suffix tree for a *set* of strings.

3. Building a Suffix Tree in linear time and space

Weiner [8] published the first algorithm for constructing suffix trees in linear time. Given the suffix tree's ability to solve complex string problems, the result that they could be constructed efficiently prompted Donald Knuth to claim Weiner's work as "the algorithm of 1973". A more space efficient version of the algorithm was given by McCreight in 1976 [9], and in 1995 Esko Ukkonen published an alternative linear-time algorithm [2]. Gusfield considers Ukkonen's to be the most conceptually simple of the three known algorithms, and my implementation was built following his description, which I summarize here.

Ukkonen's algorithm builds a sequence of intermediary structures called *implicit suffix trees* until reaching the final phase of the algorithm at which point the structure converts naturally to an 'explicit' suffix tree conforming to Definition 2.1.

3.1 Implicit Suffix Trees

An *implicit* suffix tree can be obtained from a true suffix tree by the following rules:

1. Remove every copy of the terminal symbol $ from the edge labels of the tree.
2. Remove any edge that has no label.
3. Remove any node that does not have at least two children.

For example, Figure 3.1 shows the suffix tree as has been described so far, and Figure 3.2 shows the equivalent *implicit* suffix tree for the string "xabxa$":

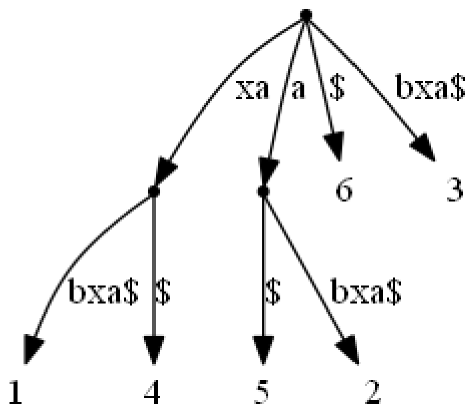


Figure 3.1: *Suffix tree for the string "xabxa$".*

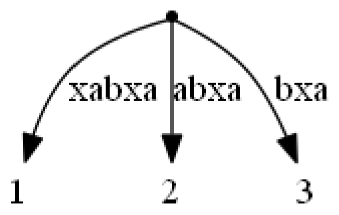


Figure 3.2: *Implicit suffix tree for the string "xabxa$".*

3.2 High Level Algorithm

At a high level, Ukkonen's algorithm proceeds from left to right along the string in a sequence of phases, constructing an *implicit* suffix tree I_i for each prefix `S[1..i]` of `S`. The true suffix tree is the final implicit tree I_n:

```
Construct implicit tree I_1
For i from 1 to n – 1 do
begin {phase i + 1}
      for j from 1 to i + 1
            begin {extension j}
            Find the end of the path from the root labelled S[j..i] in the
            current tree. If needed, extend that path by adding character
            S(i + 1), thus assuring that string S[j..i + 1] is in the tree.
            end;
end;
```

Algorithm 3.1:*High level pseudo-code of Ukkonen's algorithm for suffix tree construction, quoted from* [1].

More descriptively: "the algorithm is divided into `n` phases (one per character) and in phase `i + 1`, implicit tree I_{i+1} is constructed from the previous tree I_i. Each phase is further divided into `i + 1` extensions, one for each of the `i + 1` suffixes of `S[1..i + 1]`." [1] Implicit tree I_1 is simply the root with a single outgoing edge labelled `S(1)`, and a leaf label of `1`.

In extension `j` of phase `i + 1`, the algorithm finds the end of the path from the root labelled with substring `S[j..i]` and ensures that character `S(i + 1)` is appended to this path, in accordance with the following set of *extension rules*:

3.3 Extension Rules

Rule 1: "In the current tree, path `S[j..i]` ends at a leaf. Character `S(i + 1)` is added to the end of the label on that leaf edge." [1]

Rule 2: "No path from the end of `S[j..i]` continues with character `S(i + 1)` but at least one labelled path continues from the end of `S[j..i]`." [1]

Within this rule there are two possible cases: that path `S[j..i]` ends at a node, or inside an edge. In the former case, a new leaf edge starting at the node is created and labelled with character `S(i + 1)`. In the latter case, the edge must first be split and a new internal node created at the end of `S[j..i]`. Then, a leaf edge is added to the internal node and labelled with character `S(i + 1)`. In both cases, the leaf at the end of the new leaf edge is labelled with the number `j`.

Rule 3: Some path from the end of `S[j..i]` already continues with character `S(i + 1)`. In this case we do nothing.

As an example application of these suffix rules, Figure 3.3 shows an implicit suffix tree for the string "*axabx*" before the sixth character '*b*' is added, and Figure 3.4 shows the resulting extension:

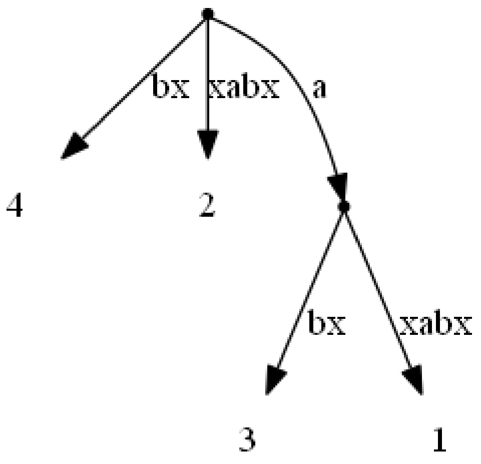


Figure 3.3: *Implicit tree for the string "axabx" before the sixth character 'b' is added.* [1]

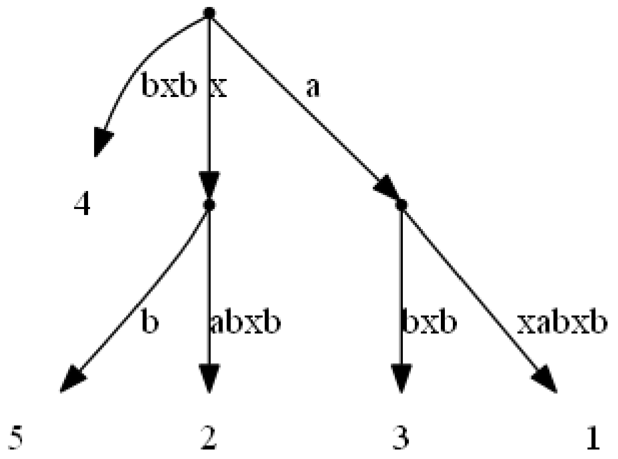


Figure 3.4: *The extended implicit tree after the character 'b' has been added. Suffixes* `S[1..n]`, `S[2..n]`, `S[3..n]` and `S[4..n]` *which correspond to path labels "axabx", "xabx", "abx", "bx" in Figure 3.3 all end at leaves so they are subject to Rule 1 and the character 'b' is simply appended to the leaf edge. Suffix* `S[5..n]` *corresponding to path label "x" in Figure 3.3 is an example of Rule 2: a pre-existing edge has been split to accommodate the new character 'b'. Finally, Suffix* `S[6..n]` *corresponding to the empty path label in Figure 3.3 is subject to extension Rule 3: there is already a path leaving the root with character 'b' so no further action is required.*

3.4 Optimizations

As currently described, the algorithm has `O(n`^{3}`)` time complexity and `Θ(n`^{2}`)` space complexity [1]; I will now describe a number of optimizations that speed the algorithm up to the desired `O(n)` time/space complexity [1].

3.4.1 Suffix Links

The first significant speed-up is achieved through the use of suffix links. A formal definition follows:

Definition 3.1: "let $x\alpha$ denote a string, where x denotes a single a character and α denotes a (possibly empty) string. For an internal node v with path-label $x\alpha$, if there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a suffix link". [1]

If α is empty, the suffix link points to the root node; we can think of the root node as having the 'empty' path label.

For example, Figure 3.5 again shows the suffix tree for string "xabxa$" (as previously shown in Figure 3.1), this time with suffix links added as dotted arrows:

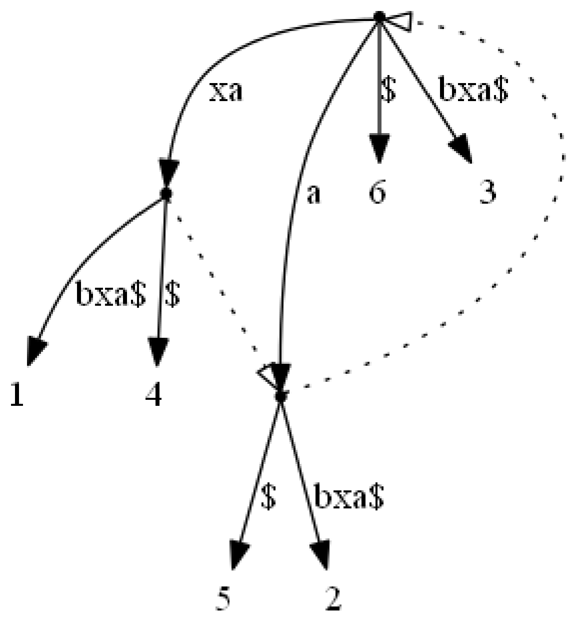


Figure 3.5: *Suffix tree for the string "xabxa$" with suffix links shown as dotted lines. There are two suffix links corresponding to the two internal nodes. The node with path label "xa" points to the node with path label "a", and the node with path label "a" points to the root (the empty path label).*

Although I omit the proof here (a detailed proof is covered in Gusfield [1]), there are two facts about suffix links that are not implied by the definition:

1. Every internal node (excluding the root) has a suffix link pointing from it to some other node. In other words, in any implicit suffix tree I_i, if internal node v has path label $x\alpha$, then there is a node w with path-label α.
2. In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

In Ukkonen's algorithm, suffix links act as a 'shortcut' to the path of the next extension. Recall from the high level algorithm that in each extension `j` of phase `(i + 1)`, we must "find the end of the path from the root labelled `S[j..i]` in the current tree". Naively, we could simply follow the path label `S[j..i]` from the root in each extension; by following suffix links we can shorten this walk considerably. Gusfield describes this optimization as the "Single Extension Algorithm":

3.4.2 Single Extension Algorithm: SEA

The first extension `(j = 1)` of any phase is special: in this extension we are appending character `S(i + 1)` to the path `S[1..i]`. This always ends at a leaf node (specifically, the first leaf node that was created in I_1), and its suffix extension is always handled by Rule 1.

Subsequent extensions `(j ≥ 2)` in phase `i + 1` are handled by the Single Extension Algorithm: SEA:

1. Find the first node `v` *at or above* the end of `S[j - 1..i]` (in other words: find the first node `v` at or above the end of the *previous extension*), that either has a suffix link from it or is the root. In any implicit suffix tree, this involves walking up *at most* one edge from the end of `S[j - 1..i]`, for a detailed proof of why this is see Gusfield [1]. The string `y` (possible empty) denotes the string between `v` and the end of `S[j - 1..i]`. Cleary, `y` will be empty only in the case that `v` is *at* the end of `S[j - 1..i]`.
2. If `v` is not the root, follow `v`'s suffix link to `s(v)` and then walk down from `s(v)` following the path for string `y`. If `v` is the root, then follow the path for `S[j..i]` as in the naïve algorithm.
3. Using the extension rules previously described, ensure that the string `S[j..i]S(i + 1)` is in the tree.
4. If a new internal node `w` was created in extension `j - 1` (which must have been due to extension Rule 2), then string α ends at a node which is `w`'s suffix link. Create the outgoing suffix link from `w` to `s(w)`.

Algorithm 3.2: *Single Extension Algorithm: SEA, quoted from* [1].

3.4.3 Walk down algorithm

The next optimization applies to the Single Extension Algorithm (Algorithm 3.2). Recall that in Step 2 of the SEA, the algorithm walks down from node `s(v)` along a path labelled `y`. A naïve implementation of this function would simply walk character-by-character, taking time proportional to the length of `y`, eg. `|y|`.

However, we can exploit a specific property of suffix trees to shorten this walk. Recall from Definition 2.1 that "No two edges leaving a node may start with the same symbol". We also know that string `y` is certainly in the tree (it is not a 'speculative' traversal). Furthermore, consider that it is trivial to find the length of any given edge.

All these facts put together allow the "walk down" process to be very efficient: instead of walking character-by-character, the algorithm works by traversing whole edges at once and keeping track of the number of characters yet to be consumed in `y`. Gusfield refers to this as the "skip/count trick":

```
Edge, Int   walk_down(Node n, String y)
{
      int h = 1;
      Edge e = outgoing edge of n starting with character y(1);
      int g = length of y;
      int g' = length of e;

      while (g' < g)
      {
            g = g - g';
            h = h + g';

            n = node at end of edge e;
            e = outgoing edge of n starting with character y(h);
            g' = length of e;
      }
      return (e, g);
}
```

Algorithm 3.3: *Pseudo-code for the "skip/count trick". The function is passed a node and a string and returns a pair of values* `e` *and* `g`*. The path following string* `y` *from node* `n` *ends at character* `g` *of edge* `e`*.*

3.4.5 Edge label compression

So far I have implied that each edge is represented by a distinct string in memory. It is also convenient and intuitive to describe suffix trees in this way. However, this is not space efficient. For example, to demonstrate that suffix trees can considerably inflate the space requirements of the original string, consider a suffix tree for the string "abcdefghijklmnopqrtsuvwxyz$":

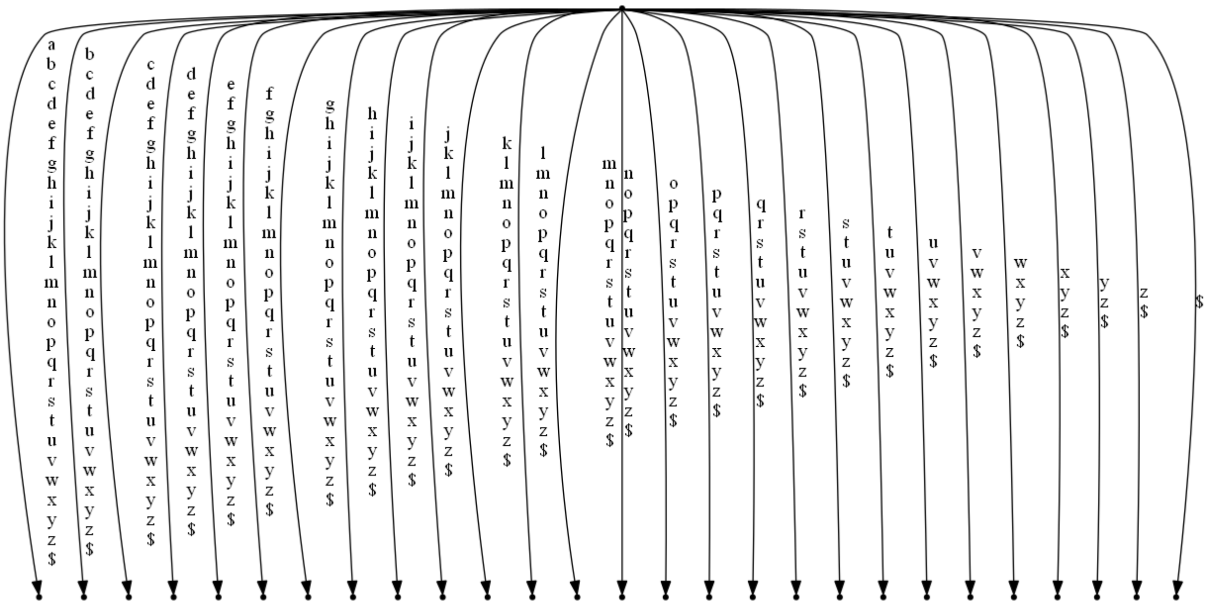


Figure 3.6: *A suffix tree for the string "abcdefghijklmnopqrtsuvwxyz$". The original string contains 27 characters; the suffix tree contains 378 characters.*

Alternatively, we can replace the string associated with each edge with a pair of indexes into the original string `S`, representing the '`start`' and '`end`' index of the contiguous characters associated with that edge. This reduces the space required to represent an edge to just two integers, regardless of the number of characters along that edge. For example, Figure 3.7 shows an example suffix tree for string "xabxa$" with the compressed edge representation:

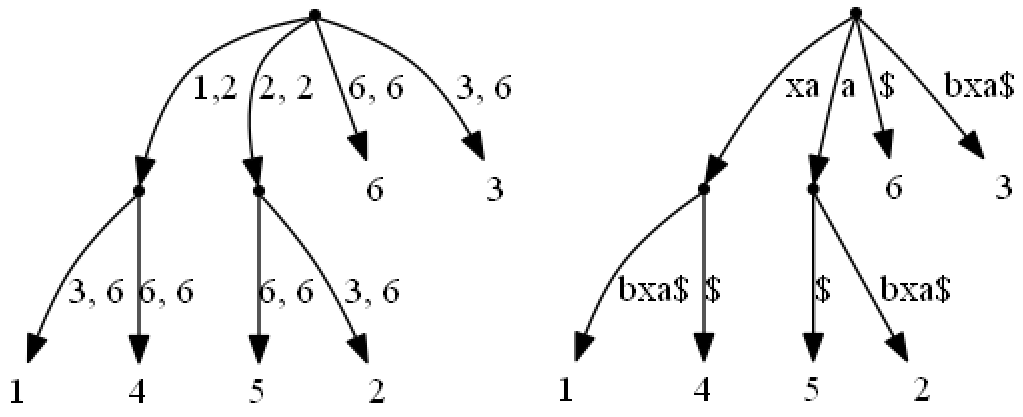


Figure 3.7: *Suffix tree representation of the string "xabxa$" with edge indexes on the left and string-based edges on the right.*

Since the number of edges in a suffix tree is *at most* `(2n - 1)` [1], this reduces the space requirements of the tree to `O(n)`. However, the string-based representation is considerably clearer for the reader and despite the underlying reality of the algorithm which represents edges with indexes, I will continue to describe and draw suffix trees in the uncompressed form.

3.4.6 Break phase after Rule 3

As per Gusfield:

"In any phase, if extension Rule 3 applies in extension `j`, it will also apply in all further extensions until the end of the phase". [1]

When Rule 3 is applied we know that the path labelled `S[j..i]` in the current tree continues with `S(i + 1)`. Consequently, we know that the path labelled `S[j + 1..i]` must also continue with `S(i + 1)` and so on for all subsequent extensions. This insight means that any phase `(i + 1)` can be ended the first time that extension Rule 3 applies.

3.4.7 Current end pointer for leaf edges

Note that in the algorithm described so far, there is no mechanism for turning a leaf into an internal node. Gusfield refers to this phenomenon as "once a leaf, always a leaf". Furthermore, any phase `i + 1` always begins with a sequence of leaf extensions subject to Rule 1 or Rule 2 (recalling from Section 3.4.6 that the application of Rule 3 ends the current phase). If we consider that j_i is the last Rule 1 or Rule 2 extension from the previous phase, then it must follow (given that any application of Rule 2 creates a new leaf), that $j_i \leq j_{i+1}$.

More simply: “The initial sequence of extensions where Rule 1 or Rule 2 applies cannot shrink in successive phases”. [1]

This insight leads us to the last optimization: when a leaf edge is created and would normally be labelled with the start and end index of its substring label, it is instead labelled with a start index and a *global current end pointer* (an integer pointer). At the start of each phase, the *global current end pointer* is incremented and this implicitly applies Rule 1 to all of the leaf edges in the tree. The algorithm then begins computing extensions explicitly at extension $j_i + 1$.

The implications of Section 3.4.6 and 3.4.7 are succinctly summarised by Tsadok [7] in his project report when he states “only Rule 2 is *real*”. In other words, the algorithm does no explicit work to apply Rule 1 or Rule 3 in any phase. This insight leads directly to the Single Phase Algorithm SPA:

3.4.8 Single Phase Algorithm: SPA

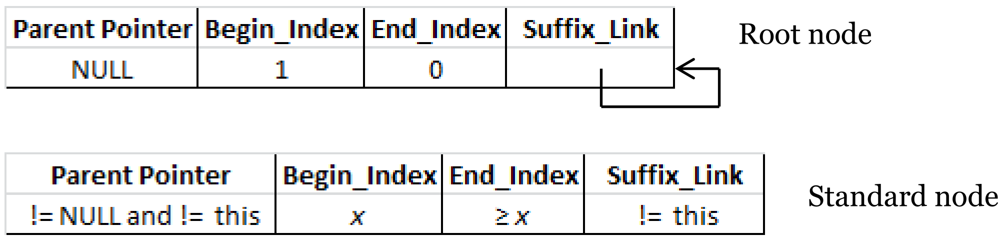
1. Increment the global current end pointer value to $i + 1$ (this implicitly computes all extensions 1 though j_i, as per Section 3.4.7).
2. Explicitly compute successive extensions using SEA starting at $j_i + 1$ until the reaching the first extension $j*$ where Rule 3 applies or until all extensions are done in this phase.
3. Set j_{i+1} to $j* - 1$ to prepare for the next phase.

Algorithm 3.4: *the Single Phase Algorithm, quoted from* [1].

Although I have omitted to prove each step, the SEA and SPA, plus the implementation tricks described finally achieves the desired $O(n)$ running time. A detailed description of the time complexity is contained in Gusfield [1] and Ukkonen’s original paper [2].

3.5 Representing Nodes and Edges

There is still a large gap between the algorithm described so far and a working program. A key design decision is to determine the appropriate representation of nodes and edges. After experimenting with several variations, I found the most sensible representation (also used by Tsadok [7]) was to have a node associated with its *incoming* edge (rather than any of its outgoing edges). This is due to the fact that every node (excluding the root) has one and only one incoming edge: there is a 1-1 relationship between nodes and incoming edges. Consequently an ‘edge’ is not a distinct class in the program design; edges are simply part of nodes. Figure 3.8 shows the canonical representation of nodes in the submitted program:



Parent Pointer	Begin_Index	End_Index	Suffix_Link
NULL	1	0	

Root node

Parent Pointer	Begin_Index	End_Index	Suffix_Link
!= NULL and != this	x	$\geq x$	!= this

Standard node

Figure 3.8: *Representation of nodes in the program. The* `begin` *and* `end` *index refer to the* incoming *edge of the node. The root node is exceptional to the standard node. Firstly, as its path label conceptually represents the 'empty string', it is the only node where its* `end` *index is less than its* `begin` *index. Secondly, it is the only node with a* `NULL` *parent pointer (having no parent). Finally its suffix link points to itself, a property which is not shared by any other node.*

In my initial design, every node had a single `sibling` pointer of type `node` and a single `child` pointer of type `node`. The `child` pointer was effectively the head of a singly linked list of the `node`'s children, with the `sibling` pointers acting as the list links. The idea of representing children as a linked list was borrowed from Tsadok's implementation [7], although in his case, a doubly linked list was used.

However, subsequently, it was necessary to change the representation of children to preserve the performance characteristics of the tree in its general form: the problem is discussed in Section 4.2.3.

4. General Suffix Trees

4.1 Description

So far we have considered Suffix Trees for one string. Now I will describe Suffix Trees for a set of strings, known as the *General Suffix Tree*. This will finally lead us onto our main application of suffix trees: solving the all-pairs suffix-prefix problem.

In the first step, we concatenate the set of strings using unique terminal symbols at the end of each string. Up until now we have used the dollar $ symbol. In the upcoming examples we will use digits instead and assume that our text consists only of alphabetical symbols (recalling that our DNA alphabet consists only of A,C,T & Gs).

For example, to build the general suffix tree for strings aba and ab, we concatenate as follows: "aba1ab2".

Following Ukkonen's algorithm builds the following suffix tree:

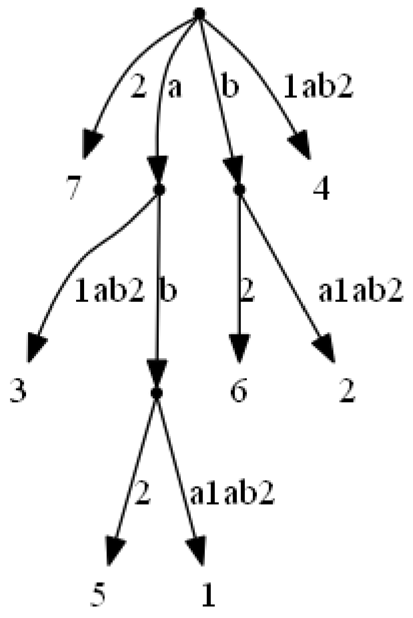


Figure 4.1: *A general suffix tree for the strings "aba" and "ab", using the digits 1 and 2 as unique terminal symbols.*

One problem with this method of construction is that we have a number of "synthetic" suffixes that span multiple strings in the set. For example, in Figure 4.1, the edges labelled "1ab2" and "a1ab2" are synthetic. A second problem is that the leaf labels are not informative: they only tell us the starting location of the path-label in the concatenated superstring. We are more interested in knowing the starting position of the path-label in the individual string S_i, and the string number `i` that the path-label relates to.

These issues can be resolved by applying two post-processing steps:

1. If an edge label contains a terminal symbol (ie. a digit in this case), remove any characters *after* the first terminal symbol of the label.
2. Label each leaf with a pair of integers consisting of the string ID and the starting position of the suffix in that string.

Algorithm 4.1: *Post-processing steps to generate a "true" general suffix tree*

Applying these rules to Figure 4.1 yields the following "true" general suffix tree:

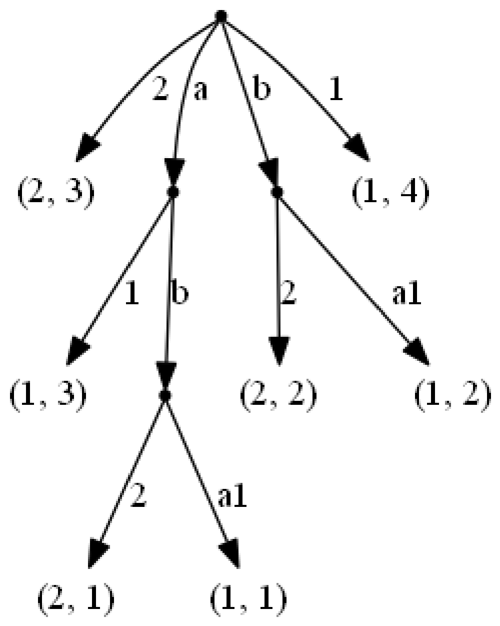


Figure 4.2: *A true general suffix tree for the strings "aba" and "ab", with post-processing applied.*

This structure allows us to solve the "exact match" problem previously described in Section 2.2, except this time we can generalize the question to a set of strings: given a pattern `P` and a set of strings S_i, find all occurrences of `P` in the set of strings S_i. In this case we expect both the string number `i` in the set and the starting position of `P` in that string; both pieces of information are encoded in the general suffix tree after the specified post-processing.

4.2 Representing General Trees

Given the theoretical approach discussed in Section 4.1, representing millions of distinct strings in a general suffix tree would require millions of distinct terminal characters. We only have a finite set of characters available: the ASCII set consisting of 128 distinct characters. The terminal character up until now has been represented by the character '$' or with digits. I found little practical information about the implementation of general suffix trees for large numbers of strings.

To work-round this problem I experimented with two different ideas:

4.2.1 Compound terminal symbols

The first idea was to introduce terminal symbols that consisted of more than one character. It turns out that the ability of suffix trees to solve the "exact match" problem is preserved with compound terminals. Assuming we decide to use digits as terminal symbols, the strings "war" and "ar" concatenated in a larger set might result in the string "war10ar11" as a substring in the tree. The general suffix tree (with post-processing) for this substring is as follows:

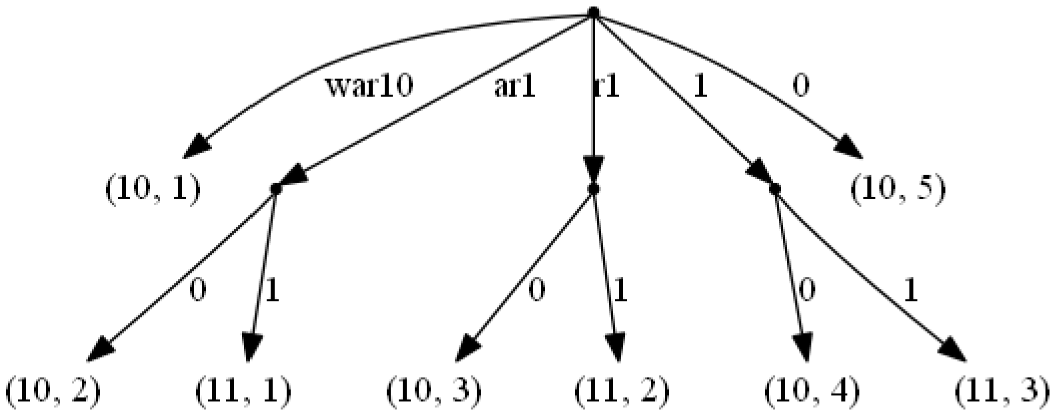


Figure 4.3: *A general suffix tree for the strings "war" and "ar" using compound terminals*

Observe that the exact match problem can be still be solved using this tree. The tree tells us that the string "ar" appears in S_{10} at position `2`, and in S_{11} at position `1`.

One issue with this approach is that compound terminals 'inflate' the size of the tree. For example, instead of using an additional one character per string to represent a million strings, if we decide to use the numbers 0 to 999,999 to represent the compound terminals it adds an additional 5,888,890 characters, this number being the count of digits in the numbers 0 to 999,999.

To improve on this we could use a number system of a much larger base (say, base-100) taking advantage of the fact that in a DNA sequencing application we need only reserve four characters for the 'non-terminal' alphabet (ie. reserving A, C, T and G and using all other characters to encode a number system). However, I decided that this would introduce unnecessary complexity into the code.

Furthermore, having initially implemented the tree with compound terminal symbols consisting of digits, I found it made other properties of the tree inconsistent with the requirements of a key algorithm which I used later in the project, and I therefore abandoned this whole approach.

4.2.2 Position-qualified terminal symbols

Exploiting the fact that no two strings end at the same position, the approach I eventually settled on was to use a single terminal symbol, '\$', qualified by its position in the concatenated string. Recall that all edge labels in the suffix tree are not explicitly represented by characters, but are represented by two integers that index into the string. Therefore, any terminal symbol character that is represented in the tree can be 'disambiguated' by its position.

Using the previous example of 'war' and 'ar', a suffix tree would conceptually be built for the following string, noting that string "war" ends at the fourth character, and "ar" ends at the seventh character:

$$\text{war\$}_4\text{ar\$}_7$$

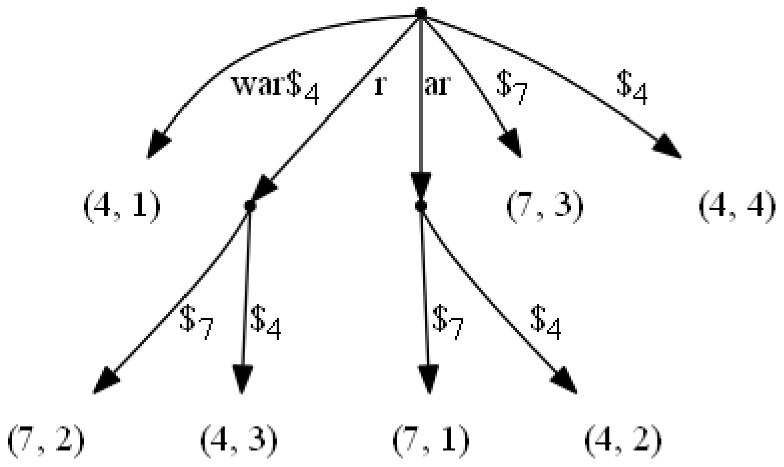


Figure 4.4: *A general suffix tree for the strings "war" and "ar" using position-qualified terminal symbols. Note the schema of the leaf labels which mirrors the identification of strings in the submitted program: a string is uniquely identified by the position of its terminal symbol.*

For a moderate increase in code complexity, this idea enabled general suffix trees to be built for an arbitrarily large number of strings. The additional code complexity was introduced whenever a decision in the algorithm had to be made on the basis of specific characters matching or not matching. For example, to test if a node has a child beginning with a certain character it was no longer sufficient to simply compare the characters. Instead, an exception case would have to be made if the symbol was a terminal, in which case the algorithm would resort to comparing the relative position of the symbol in the concatenated string.

4.2.3 Representing children with the C++ map container

Up until now, in the context of DNA sequencing, the alphabet used within the suffix tree has been fixed: $\sum$ `= {A, C, T, G, $}`. This means any node in the tree could have, at most, five children. However, the introduction of position-based terminal symbols means that the alphabet has become arbitrarily large to accommodate all the possible terminal symbols. I ran tests on the program for a set of 10,000 36-character strings, composed of 'A','C','T' and

‘G’ characters extracted from the Swinepox virus DNA sequence [11]. Seventy-two nodes had one-hundred or more children; eight nodes had one-thousand or more children and the ‘worst’ node had 8632 children. This phenomenon created severe performance degradation as the size of the string set increased. In fact, the algorithm lost its `O(n)` properties in the *general* suffix tree form and began to take progressively longer to complete each phase of Ukkonen’s algorithm.

To isolate the problem, I used Visual Studio’s inbuilt profiling tools to compare a run of the program for a single string, and a set of strings with the same *total* length. This was done using the ‘Performance Wizard’ in the ‘Analyze’ menu. The profiles were performed using “CPU Sampling”, which is a technique that gathers statistics about CPU usage. One of the output reports from CPU Sampling is a breakdown of % CPU-time spent in each function of the program. The most costly function in each case was the `get_child` member function of the Node class.

Function Name	% CPU time in non-general suffix tree	% CPU time in general suffix tree
`Node::get_child(class SuffixTree const &,int)`	6.45	62.4

Figure 4.5: *Comparing the % CPU time spent in the get_child function for non-general and general suffix trees.*

As can be seen from the table, the `get_child` function was significantly more costly in the general suffix tree (almost 10 times as costly for the dataset used). The `get_child` function returns the child of a node where the outgoing edge to that child node starts with a specified character. The character is passed to the function in the form of an integer `i`; the function looks up the character `S(i)` from the suffix tree `T`. If there is no outgoing edge starting with the correct character the function returns `NULL`.

It was now clear why the position-based terminal symbols were causing degradation in performance: representing the children as a linked list meant that the algorithm might have to scan through the entire list to find the correct child. Gusfield [1] highlights this as a potential problem with Suffix Trees that have large alphabets and recommends sorting the linked list to try and reduce the amount of time spent searching (the search can be abandoned once the theoretical position in the list is reached). However, this approach did not achieve a significant gain in efficiency. It also added a lot of code complexity when adding children to nodes, and when removing children of nodes (as happens in the case of a split edge).

After investigating several other ways of representing children of nodes (including the C++ `vector`), the approach I settled on was to use the inbuilt C++ `map` associative container. Maps store a key value (of any type) and associated mapped value (of any type). The performance of various operations on `Map` containers according to the ISO C++ standard [12] are as follows:

Operation	**Performance**
Insertion	O(log n)
Lookup	O(log n)
Deletion	O(log n)

Figure 4.6: *Performance characteristics of the C++* `map` *container.*

Since the algorithm needs to insert, lookup and delete children, these performance characteristics provide a good balance between all three operations. Furthermore, using the inbuilt C++ container class made for extremely clear and succinct code.

In actual implementation, it was clear that the *mapped* value should be the child node itself, but the correct *key* value required some thought. The first character of the outgoing edge is the 'obvious' choice of key, but is not sufficient: recall that (as per Section 4.2.2) terminal symbols are uniquely represented by their *position* in the string rather than the character itself. There are two different requirements: on the one hand, for non-terminal symbols the position in the string is irrelevant and the key can be a simple `char`, on the other hand terminal symbols must differentiated with an integer which (as the last character of a tree is always a terminal symbol) may be as large as the maximum length of the `string`-type in C++.

To reconcile these two different cases, I used an integer as the `map` key and took advantage of the fact that the position of a terminal symbol would always be a *positive* integer. This left all the *negative* integers free to represent non-terminal symbols. Non-terminal characters were converted into negative integers by multiplying the character number by -1. For example:

Character	**Mapped Integer**
A *(ASCII character 65)*	-65
C *(ASCII character 67)*	-67
T *(ASCII character 84)*	-84
G *(ASCII character 71)*	-71

Figure 4.7: *Non-terminal characters at the start of edges are converted to negative integers, which become the* key *for the child node.*

Terminal symbols are simply represented as *positive* integers corresponding to their position in the suffix tree string. For example, a '$' symbol at position 65 is represented by the integer 65.

4.3 Solving the all-pairs suffix-prefix problem

The overlap between each string in a set of strings is a vital input to the overlap graph method for de novo sequence assembly described in Section 5. General suffix trees solve this problem efficiently.

The general suffix tree must be augmented with additional information in order to compute pairwise overlaps. Firstly, the node-depth of each node must be known. This can be calculated as the algorithm computes the overlaps and does not need to be stored explicitly. However, the definition of "node depth" in this context is slightly different to the standard tree definition: the "node depth" of a node in a suffix tree is the *length of the path label of* *that node,* where the path label is the concatenation of all edge labels leading to the node from the root. Secondly, some additional labels must be applied to certain internal nodes in the tree. Specifically:

"Label each inner vertex `x` of the suffix tree with a subset $L(x) \subseteq \{1, \ldots, N\}$, such that $i \in L(x)$ holds if and only if `x` is incident to an edge with label $\$_i$".

Algorithm 4.1: *Pre-processing nodes in order to solve the all-pairs suffix-prefix problem (quoted from [4]).*

Nodes can be labelled either during construction of the suffix tree, or afterwards using a standard depth-first search to traverse the tree structure. For the sake of code simplicity, I chose the latter option.

Figure 4.8 shows an example of node labels for the strings "aba", "bab" and "aabb", concatenated as "aba$\$_4bab\$_8$aabb$\$_{13}$", with each string uniquely identified by the position of its terminal symbol:

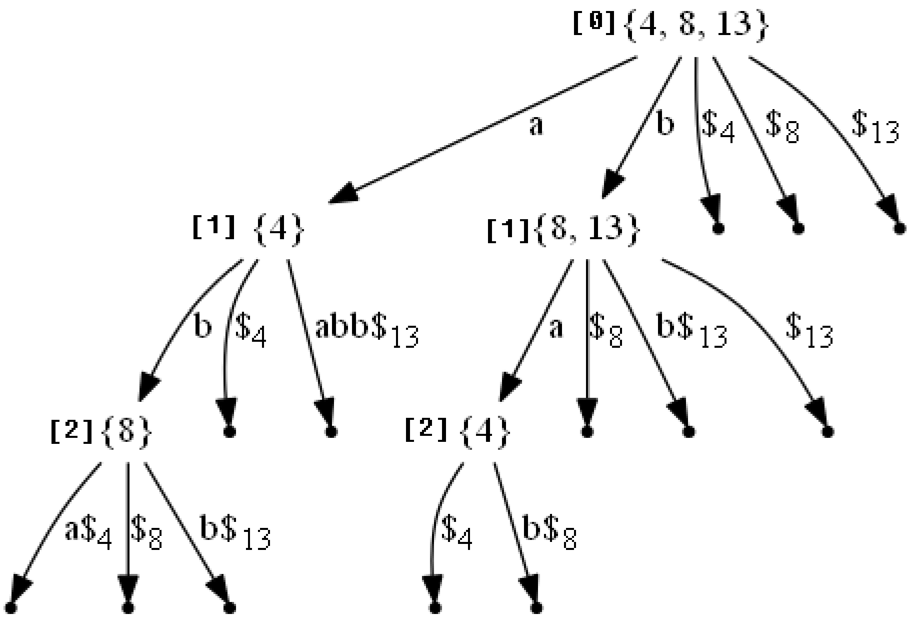


Figure 4.8: *Node labelling for the suffix tree "aba$\$_4bab\$_8$aabb$\$_{13}$". Node depth is shown in square brackets, node labels are shown in curly brackets. For simplicity, leaf labels have been removed from the diagram as they are not relevant for computing overlaps. In this case, internal nodes have path labels consisting of one-character edges so node depths happen to corresponds with the usual definition of node depth in a tree. This would not necessarily be the case in a more complex example.*

To see how node labels and node depth can be used in tandem to compute pairwise overlaps, we will now run through an example: computing the suffix-prefix match between S_i, S_j where S_i = "aba" (ending at terminal position 4 in our set) and S_j = "bab". The general method is to follow the path-label for S_j (ie. the string which is having its prefix matched), recording the maximum node depth at which a node is labelled with the terminal position of S_i (in this case, the label `{4}`). In Figure 4.8, the deepest node with label `{4}` on the path of "bab" has a depth of [2]. Therefore the suffix-prefix match between "aba" and "bab" is 2 (eg. "ba").

This idea can be easily extended to compute all pairwise suffix-prefix matches. The algorithm would follow the path for each string S_j and record the maximum node depth encountered for each S_i terminal symbol on the path of S_j.

5. Overlap Graphs

5.1 Description

Now that we have shown (in Section 4.3) an efficient way of computing pairwise suffix-prefix overlaps between a set of strings, we are ready to describe the overlap graph and its use in solving the SCS problem.

A formal definition follows, quoted from [4]:

Definition 5.1: Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet Σ. The overlap graph $G_{ov}(S)$ is the complete edge-weighted directed graph $G_{ov}(S) = (V, E, c)$, where

1. $V = S$ and $E = V^2$, and
2. $c : E \rightarrow N$, with $c(s_i, s_j) = ov(s_i, s_j)$ for all $s_i, s_j \in V$. In other words: an edge that connects two strings is labelled by the suffix-prefix overlap length between those two strings.

This definition includes "self-loops", eg. the suffix-prefix overlap of a string with itself, or $ov(s_i, s_j)$ where $i = j$. However, in the context of de novo sequence assembly, an individual read cannot overlap with itself so from this point onwards we will not consider "self-loops" to be valid candidates for overlap.

The following diagram shows an overlap graph for the set of strings “caba”, “ababaa”, “aabca”, “aacab” and “aaddd”:

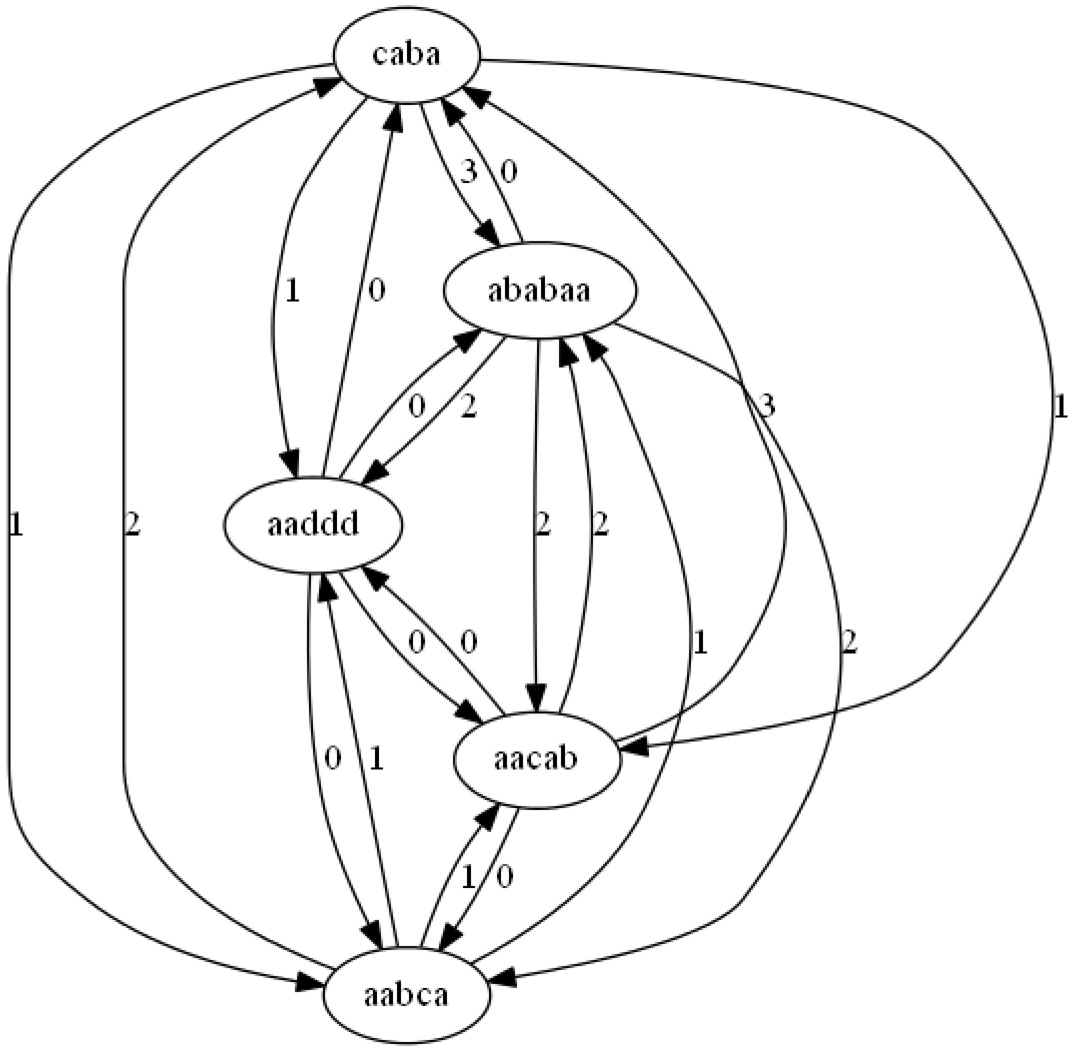


Figure 5.1: *an example overlap graph for the strings “caba”, “ababaa”, “aabca”, “aacab” and “aaddd”, duplicated from* [4].

Note that the overlap graph is never constructed explicitly as a data structure in my implementation. The pairwise overlaps between strings contain all of the information needed to manipulate the overlap graph structure.

Conceptually, how is the overlap graph used to find solutions for the SCS problem? Any route through the graph which visits each node exactly once is a candidate superstring for the set of strings. However, the *shortest* string is the route through the graph that maximizes the edge weights. This is similar to the travelling salesman problem, except in this case we are trying to maximize the sum of the edge weights rather than minimize them.

To put the ‘hardness’ of this problem into perspective: the largest travelling salesmen problem that has currently been solved is 85,900 cities [13]. Our string sets in biological applications, especially for de novo sequence assembly are likely to be in the millions, if not billions.

At this point it should be clear why the SCS problem is a NP-hard optimization problem, and why, in practice, heuristic or approximation methods are almost always used to solve the SCS problem for large string sets.

5.2 The Greedy Superstring Algorithm

We are now ready to introduce the algorithm which creates the superstring in my implementation: the greedy superstring algorithm. The greedy superstring algorithm is an approximate solution for the SCS problem:

```
Input: A set of strings $S = \{s_1, \dots, s_n\}$

while (size of S > 1) do

    1. Find $s_i, s_j \in S,\ s_i \neq s_j$, that have a maximum overlap among all strings in S
    2. Let $s' = \langle s_i, s_j \rangle$ be the merge of strings $s_i$ and $s_j$.
    3. Delete $s_i, s_j$ from S and insert $s'$ into S.

Output: The only remaining string $s_{greedy} \in S$.
```

Algorithm 5.1: *Greedy Superstring algorithm, duplicated from [4].*

	ababaa	**caba**	**aaddd**	**aabca**	**aacab**
ababaa	N/A	0	2	2	2
caba	3	N/A	1	1	1
aaddd	0	0	N/A	0	0
aabca	1	2	1	N/A	1
aacab	2	3	0	0	N/A

Figure 5.2: *Pairwise overlaps between strings in Figure 5.1, duplicated from* [4].

For example, using the set of strings in Figure 5.1, and the table of pairwise overlaps in Figure 5.2, the algorithm would proceed as follows:

```
Initial set: {caba, ababaa, aaddd, aabca, aacab}
1st Merge: {cababaa, aaddd, aabca, aacab}
2nd Merge: {aacababaa, aaddd, aabca}
3rd Merge: {aacababaaddd, aabca}
4th Merge: {aabcaacababaaddd}

Therefore Sgreedy = aabcaacababaaddd
```

5.2.1 Naïve-implementation

Although the conceptual description of the greedy algorithm is very simple, an efficient implementation is slightly more complex. However, I will first briefly describe a naïve method for implementing the greedy merge, given by [4], which runs in $O(n^2)$ time complexity (where n is the number of strings in set S).

The naïve approach uses a matrix A of n^2 entries, corresponding to pairwise overlaps between a set of strings (ie. Figure 5.2).It proceeds by creating a list L of objects consisting of overlaps (expressed as integers) and pointers to the corresponding entry in the matrix A, ie "providing us with the pair of strings which corresponds to this overlap" [4]. The list L is then sorted using a "counting sort" (described in [4]). This sort proceeds by counting the number of distinct key values and using basic arithmetic to determine the position of each key in the output. It is particularly fast for sorting integers that lie in a small range (as in the case of overlaps).

We now have the maximum overlapping pair of strings at the top of the list L, (eg. Step 1 in Algorithm 5.1). Step 2 is trivial: we simple merge the strings s_i and s_j into s'. Step 3 is more involved, we need to remove s_i and s_j from matrix A and insert s'. Furthermore, any overlaps in L with pointers to s_j or s_j must be removed, or updated so that they point at s'.

Actions performed on the matrix A are as follows:

"$A[s', t] = A[s_j, t]$, $A[t, s'] = A[t, s_i]$ and $A[s', s'] = -1$, for all t in the current set S without s_i, s_j, and s'. Here, we set $A[s', s'] = -1$ to guarantee the condition $s_i \neq s_j$ in Step 1. The rows and columns corresponding to s_i and s_j are removed from A". [4]

Finally, actions performed on list L are as follows:

"Remove its first entry and rearrange pointers to A. If there was a pointer to $A[s_j, t]$, set it to $A[s', t]$, and if there was a pointer to $A[t, s_i]$, set it to $A[t, s']$. Furthermore, remove all elements from L that point to entries $A[s_i, t]$ and $A[t, s_j]$". [4]

Steps 1, 2 and 3 are iterated until S contains only one string, the greedy SCS for set S.

5.2.2 An efficient implementation: Turner's algorithm

My initial implementation was the naïve version of the greedy algorithm described in Section 5.2.1. However, after running some tests on large data sets it was clear that $O(n^2)$ was not fast enough to be competitive with existing de novo assembly software. To improve performance, I implemented a faster version of the greedy algorithm, given by Turner [3].

The main insight into a more efficient implementation is that the naïve algorithm spends most of its time running through the sorted list of overlaps to reflect changes to the set of strings. After each merge, the algorithm runs through the complete list of overlaps, even though we know that only a few overlaps will need to be updated: those that point to either of the two strings that were just merged. Conceptually, we would prefer to avoid this laborious process and react only to 'invalid' overlaps in the sorted list as we encounter them.

To accomplish this we will make use of the general suffix tree structure that was constructed for the calculation of overlaps, but we must first define two additional 'helper' functions for a suffix tree T of a set of strings S:

1. `SuffixTree.delete(i)` function. It simply uses lazy deletion to maintain an array of Boolean values corresponding to whether the string associated with integer i is deleted.

2. `SuffixTree.lookup(i, j)` function. According to Turner [3]: “Returns a pair `[l, k]`, where `l` is the length of the longest prefix of s_i which is also a suffix of some string in $S - \{s_j\}$ and s_k is one such string”. It’s important to note that this function takes the deletion status into account: it will not return s_k if s_k is deleted. The exclusion of s_j from the potential return values is necessary because we need to avoid attaching a partial solution to itself (this will be explained shortly).

   To make this operation more efficient, when a lookup for string `i` is *first* performed (using the technique in Section 4.3), a pointer to the node that corresponds with the path-label of the overlap is stored. On subsequent lookups for string `i`, the function starts from the stored pointer (instead of traversing the tree from the root) and updates the pointer if necessary: if there are no available overlaps at the current node, if traverses up to the node’s parent and so on until a suitable overlap can be returned.

Each string `i` is stored with four mapped values: `left(i)`, `right(i)`, `leftend(i)` and `rightend(i)`. As per Turner [3]:

“The algorithm does not explicitly combine strings, but keeps track of the decisions made using the two mappings `left(i)`, `right(i)` which give the left and right neighbours of string `i` in the solution constructed so far. The solution is returned in these mappings. If a string `i` has no left neighbour yet, `rightend(i)` is the original string which is currently rightmost in the piece of the partial solution that contains `i`; `leftend(i)` is similar”. [3]

This idea is represented visually below:

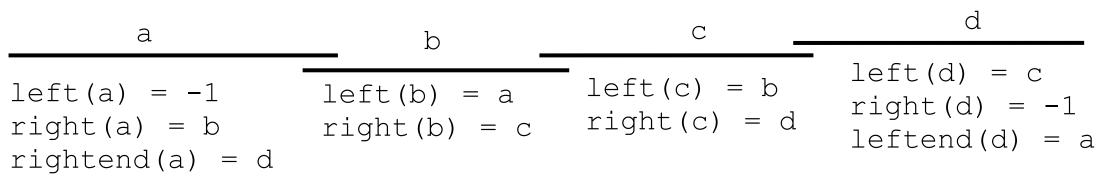


Figure 5.3: *Visualization of the string mapping schema used by Turner’s algorithm. The diagram shows a partial solution for the strings* `a`*,* `b`*,* `c` *and* `d`*. The value ‘*`-1`*’ is used as a placeholder value to mean the string is not yet merged with a neighbour on its right or left side. String* `a` *has no left neighbour, but it has a* `rightend` *value corresponding with the rightmost string in the partial solution. Similarly, string* `d` *has no right neighbour but a* `leftend` *value corresponding with the left most string in the partial solution.*

Now it should be clear why the `SuffixTree.lookup` function takes an additional parameter `j` which is excluded from the potential return values. Using the example in Figure 5.3, imagine we are looking for string `a`’s left neighbour. Only strings that do not yet have right neighbours are candidates. Note that `d` fulfils this criterion: it has not yet been merged on its right side. If we assume that `d` is in fact the string with the largest suffix-prefix overlap with `a`, this would lead to the merging of string `a` and string `d`. However this leads to a ‘dead-end’: a partial solution which loops back on itself.

Therefore, in this case we would pass string d (the rightmost string of the partial solution containing a), as the second parameter in the lookup function, ensuring that the function returns the longest overlapping string *other* than d.

In contrast to the naïve algorithm, which stores *all* the pairwise suffix-prefix overlaps in a 2D matrix, Turner's algorithm only stores the *maximum* suffix-prefix overlap for each string in a heap, using the overlap length as the heap's key. Therefore, n overlaps are recorded for n strings as opposed to n^2 in the naïve method. A heap is used to enable efficient retrieval and removal of the current string i with the longest prefix overlap, and efficient insertion of new overlaps. The heap H has three functions, insert(), get_max() and delete(i):

1. Heap.insert(l, i): Inserts i into the heap, using l as the heap key, where S_i has a prefix overlap of length l.
2. Heap.get_max(): Returns a pair [k, i], where S_i is the string with the longest prefix overlap in the heap, and k is the length of that overlap.
3. Heap.delete(i): Deletes string S_i from the heap.

With this background, we are now ready to present the efficient algorithm. This is Turner's algorithm, although the presentation is simplified to emphasize the important conceptual aspects: the original version can be found in [3]:

Input: a general suffix tree T for the set of strings S_n, a heap H of the maximum prefix-overlap for each string.

```
while (|H| > 1) {
      [k, i] ← H.get_max();
      H.delete(i);
      [l, j] ← T.lookup(i, rightend(i));
      if (l = k) {
            left(i) ← j;
            right(j) ← i;
            leftend(rightend(i)) ← leftend(j);
            rightend(leftend(j)) ← rightend(i);
            T.delete(j);
      }
      else {      //eg. l < k
            H.insert(l, i);
      }
}
```

Output: a set of string mappings corresponding to the greedy SCS.

***Algorithm 5.2:** Turner's greedy merge algorithm* [3]. *Note that S_j is merged to the left of S_i, hence* left(i) = j *and* right(j) = i. *The algorithm proceeds by retrieving the string S_i with the longest overlap from the heap, it also removes S_i from the heap. It then checks whether the overlap is still valid in the suffix tree. If so the mappings for S_i, S_j are updated and S_j is deleted from the suffix tree (it is no longer available to have its suffix matched). Otherwise S_i is inserted back into the heap.*

Turner gives the running time of this algorithm as O(m log n), where m is the total number of characters in the string set S_n [3].

6. Testing

6.1 Algorithm correctness

To test the construction of basic suffix trees I found it extremely helpful to compare the structure of the output visually with the expected tree. I accomplished this with the open source graph visualisation software ‘Graphviz’ [17]. I will briefly talk about the tree logging features of the program.

6.1.1 Graphviz logging

The Graphviz software takes basic textual information describing a graph structure as an input and produces a visualisation of the graph that can be saved and displayed as an image. The input is provided to the program in the form of a text file with the `.gv` extension.

Graphs are described using the Graphviz DOT language. The description is enclosed within curly brackets and labelled with a graph type and label. In this case the type is a directed graph which is known as a ‘digraph’ in the DOT syntax:

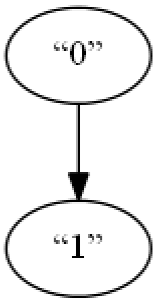
```
digraph g {
      //graph description goes here
}
```

Each node in the graph must be associated with a unique ID. This allows the DOT interpreter to determine the correct ownership of edges when nodes have multiple children. For this I re-purposed the Node class member variable “ID”. The ID is assigned to the Node at run-time. For leaf nodes, the ID reflects the leaf label, recalling from Section 2.1 that the path-label of the root to leaf `i` represents the suffix of string `S` that starts at position `i`. For internal nodes, the assigned ID is essentially meaningless and is only included to differentiate the node for logging purposes: I used a decreasing series of negative numbers to label the internal nodes so that there was no overlap with any leaf labels (which are all positive integers).

To draw two nodes connected by an edge in the DOT language we can write the following:

```
digraph g {
      “0” -> “1”;
}
```

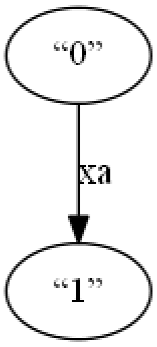
This would draw the following graph in Graphviz:



To label the edge requires an additional clause:

```
digraph g {
     "0" -> "1" [label = "xa"];
}
```

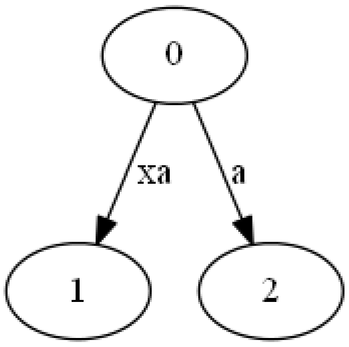
Which results in the following graph:



As each node is represented by a unique ID, drawing two children from the same node is accomplished simply with an additional line in the description:

```
digraph g {
      "0" -> "1" [label = "xa"];
      "0" -> "2″ [label = "a"];
}
```

Which results in the following graph:



This programmatic method for drawing graphs was very useful for debugging and to visually inspect the suffix tree structure for validity. The `log_tree` member function of the SuffixTree class calls the helper function `log_node` which recursively logs `parent->child` relationships and edge labels until the entire tree has been traversed. The final logged tree is written to a file called “`tree.gv`” in the program root directory. This file can immediately be opened in Graphviz for inspection.

For example, here is the DOT code and suffix tree constructed by the program for the string “xabxac”, which is an example from Gusfield on page 91 [1]:

```
digraph g {
     "0" -> "-1" [label = "xa"];
     "-1" -> "4" [label = "c"];
     "-1" -> "1" [label = "bxac"];
     "0" -> "6" [label = "c"];
     "0" -> "3" [label = "bxac"];
     "0" -> "-2" [label = "a"];
     "-2" -> "5" [label = "c"];
     "-2" -> "2" [label = "bxac"];
}
```

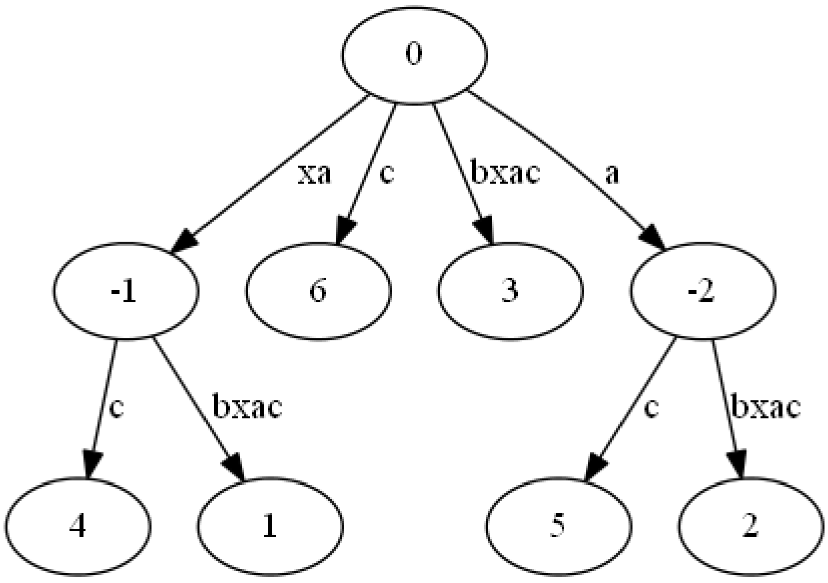


Figure 6.1: *Program output for the string “xabxac”.*

Figure 6.1 visually matches the diagram in Gusfield’s example. Also note that it satisfies the definition of a Suffix Tree in Section 2.1. Of particular note is that the path label from the root to leaf `i` reflects the `suffix(i)` of “xabxac” starting at position `i`:

```
suffix(1) = xabxac
suffix(2) = abxac
suffix(3) = bxac
suffix(4) = xac
suffix(5) = ac
suffix(6) = c
```

Note: although the logging function also works for *general* suffix trees, it does not exactly reflect the definition of the general suffix tree given in Section 4.1. In Section 4.1, the general suffix tree is shown with its leaf edges ‘tidied’ to remove synthetic suffixes. However, in the application of general suffix trees for greedy assembly (eg. Algorithm 5.2) I found this step was superfluous to requirements so the post-processing in Section 4.1 became redundant. Furthermore, the logging function does not display terminal symbols qualified with their position so the presentation is misleading for the user.

6.1.2 Testing larger Trees

Obviously the visual approach is not scalable for larger trees. To test larger trees, I built the suffix tree for a much longer string and checked that leaf IDs below a certain path label corresponded with starting positions of that path label in the string. This confirmed the 4th property of suffix trees in Section 2.1: that path labels from the `root` to `leaf(i)` correspond with suffixes of the string starting at position `i`.

The long string used was the DNA sequence of the Swinepox virus downloaded from the National Centre for Biotechnology Website [11]. This is a 146454-character long sequence composed of A, C, T & Gs. The test substring was "TGTAACCT", which appears at positions 138, 36082 and146447.

The test program resides in a file called `TestSuite.h` and takes the form of a function called "`SUFFIX_TREE_TEST`". This function returns a Boolean value depending on whether the correct leaf labels are returned by the "`get_exact_matches`" member function of the Suffix Tree class.

6.1.3 Solving a small SCS problem

The `TestSuite.h` file also contains a test for a small SCS problem consisting of the strings "ababaa", "caba", "aaddd", "aabca" and "aacab". This is a duplicate of a toy example found in [4]. The function "`GREEDY_SCS_TEST`" returns a Boolean value of `true` if the greedy SCS algorithm returns the string "aabcaacababaaddd", ie. one of the shortest common superstrings for this set.

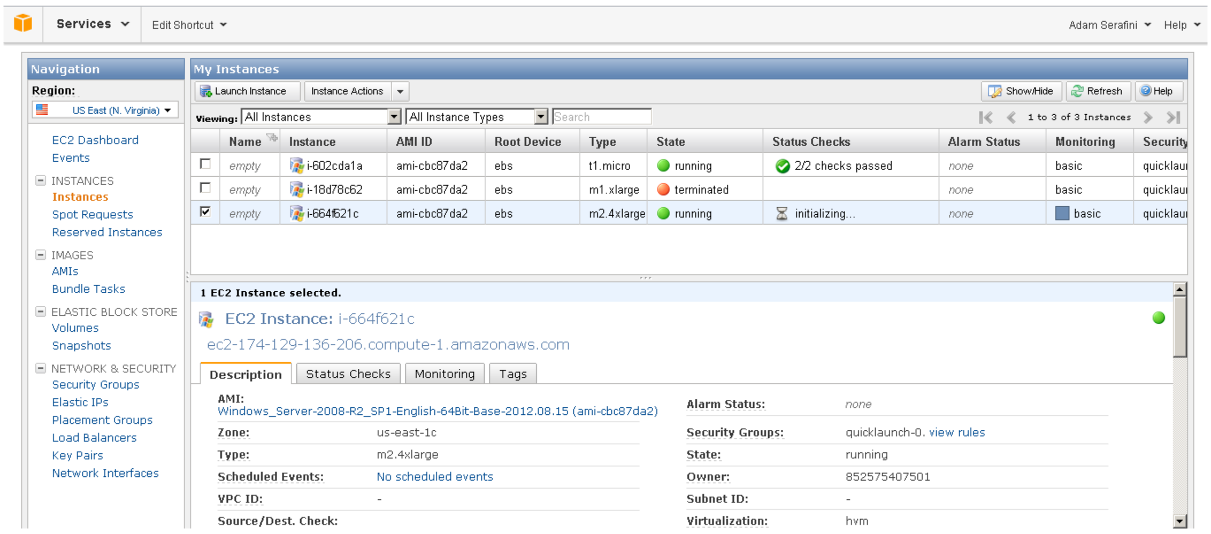
6.2 Solving a large SCS problem

I used Amazon's network of high performance cloud computers to test the program on a reconstruction of the Swinepox virus found at [11]. Two Amazon services were used: the Elastic Compute Cloud (EC2) and Elastic Block Store (EBS).

6.2.1 Elastic Compute Cloud (EC2)

EC2 is the 'heart' of Amazon's cloud service and is the basic unit of computation available on the service. An individual cloud computer is referred to as an EC2 "instance". Instances are available running either the Linux/UNIX or Windows operating systems (although Windows instances are slightly more expensive). Usage of an instance is charged by the hour. Despite the extra cost, I decided to use Windows instances. This avoided any porting issues that could have arisen in Linux, given that the code was originally written and tested on the Windows platform.

Although instances can be launched dynamically using an API, Amazon also provide a web-based "Management Console" which enables the user to launch instances and manage the periphery services for each instance. I used the Management Console to launch an instance: in the cloud jargon I "provisioned" an instance. This was much easier than interacting with the API directly; the API is really designed for dynamic provisioning and deprovisioning in response to increases or decreases of workload. In this case I was only launching and testing on a single machine with a fixed workload that was known in advance.



	Name	Instance	AMI ID	Root Device	Type	State	Status Checks	Alarm Status	Monitoring	Security
☐	empty	i-602cda1a	ami-cbc87da2	ebs	t1.micro	● running	✓ 2/2 checks passed	none	basic	quicklau
☐	empty	i-18d78c62	ami-cbc87da2	ebs	m1.xlarge	● terminated		none	basic	quicklau
☑	empty	i-664f621c	ami-cbc87da2	ebs	m2.4xlarge	● running	initializing...	none	basic	quicklau

1 EC2 Instance selected.

EC2 Instance: i-664f621c

ec2-174-129-136-206.compute-1.amazonaws.com

AMI:	Windows_Server-2008-R2_SP1-English-64Bit-Base-2012.08.15 (ami-cbc87da2)	Alarm Status:	none
Zone:	us-east-1c	Security Groups:	quicklaunch-0. view rules
Type:	m2.4xlarge	State:	running
Scheduled Events:	No scheduled events	Owner:	852575407501
VPC ID:	-	Subnet ID:	-
Source/Dest. Check:		Virtualization:	hvm

Amazon Web Services Management Console.

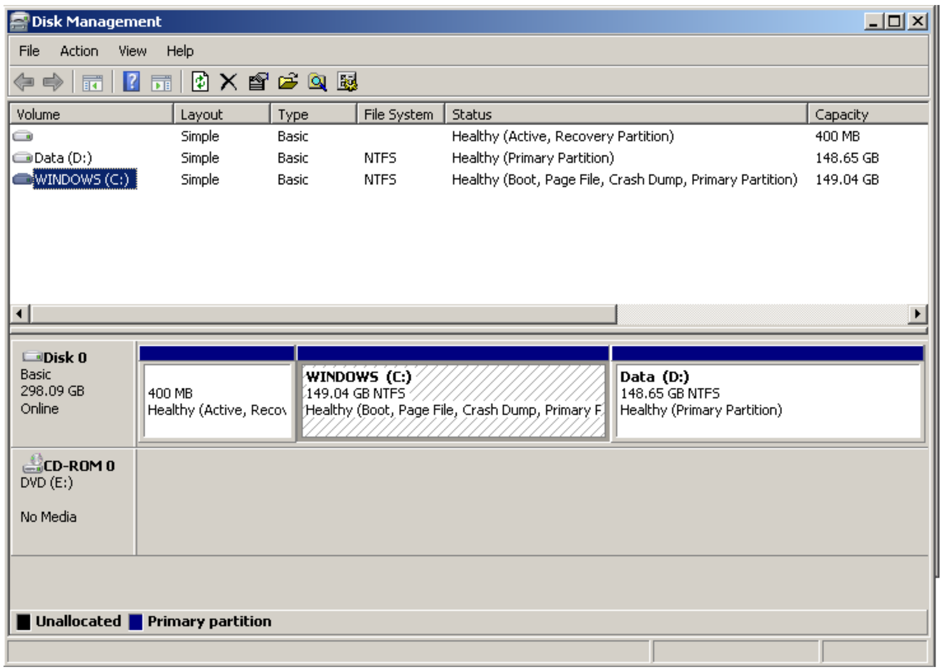
Amazon uses the Secure Shell (SSH) network protocol and public-key cryptography to authenticate users accessing the instance remotely. After I created a key pair in the management console and downloaded the corresponding private key file, the file was then uploaded against a launched Windows EC2 instance, also in the Management Console, and I was able to retrieve the Administrator password for that machine.

The Windows instance can be accessed remotely through the Remote Desktop Connection application that is available in any installation of Windows 7. Launching the Remote Desktop Connection presents the user with a dialog box requesting the Public DNS of the remote machine; the Public DNS of the Amazon instance is available in the Management Console. From this point onwards, the process was identical to operating a Windows PC. I was presented with a dialog box requesting my login username (in this case: 'Administrator') and the Windows password that was previously retrieved from the Management Console.

6.2.2 Elastic Block Store (EBS)

It is important to note that an Amazon EC2 instance is strictly "virtual" and does not correspond with a physical machine in Amazon's data centre. At an Amazon data centre, powerful servers are partitioned into any number of virtual servers "running their own operating systems in their own allocated memory, CPU and disk footprints" [19]. This introduces the question: what happens to information stored on the virtual hard disk of an EC2 instance when it is terminated? In fact, this data is lost. Furthermore, to move data in and out of the AWS network has a small cost associated with it. Therefore it was desirable to have data persist in the AWS network after my instance was terminated. This would also enable experimentation: by attaching the data set to different instance types (with various RAM and CPU specifications) I would be able to use the various instances without the inconvenience of transferring raw sequence data into the machine each time.

The Amazon solution to this use-case is the Elastic Block Store (EBS). Again, setup was accomplished using the AWS Management Console. Blocks of storage (called "volumes") ranging from 1 GB to 1 TB in size can be provisioned and attached or de-attached from specific instances. Once attached, I had to "mount" the EBS volume as a drive in the Windows instance. After logging in to the instance, I used the Windows Disk Management utility tool to perform the mount. The utility can be accessed by typing "`diskmgmt.msc`" at the Windows command prompt (`cmd.exe`):



Volume	Layout	Type	File System	Status	Capacity
	Simple	Basic		Healthy (Active, Recovery Partition)	400 MB
Data (D:)	Simple	Basic	NTFS	Healthy (Primary Partition)	148.65 GB
WINDOWS (C:)	Simple	Basic	NTFS	Healthy (Boot, Page File, Crash Dump, Primary Partition)	149.04 GB

```
Disk 0          400 MB                     WINDOWS (C:)                            Data (D:)
Basic           Healthy (Active, Recov     149.04 GB NTFS                          148.65 GB NTFS
298.09 GB                                  Healthy (Boot, Page File, Crash Dump, Primary F   Healthy (Primary Partition)
Online

CD-ROM 0
DVD (E:)

No Media
```

■ Unallocated ■ Primary partition

Windows 7 Disk Management Utility.

If the EBS volume has been correctly attached to the instance in the Management Console, it will appear in the Windows Disk Management Tool. It can be mounted by right-clicking on the drive in the disk management utility and selecting "Online".

If an EBS volume is being used for the first time, it will also need to be formatted with a file system before it can be used by the Windows instance. This was accomplished easily in the Disk Management tool by right clicking on the volume and selecting "Initialise Disk" whereupon the user is presented with a dialog box allowing them to choose a partitioning scheme.

Once these steps were completed the volume appeared as the D:\ drive of the Amazon EC2 instance. Furthermore, any data stored in the volume would persist even after the specific instance was terminated, and I could detach the volume in the management console and re-attach it to a different Windows instance with different specifications if I wished.

6.3 Constructing the Swinepox virus

At this stage I was ready to test if a sequence could be reconstructed from its substrings. I again used the Swinepox virus downloaded from the NCBI website [11]. To simulate the 'shotgun sequencing' process I wrote a small utility program that took a file containing the original 146454-character long string as an input and produced an output file containing the set of all 141-character long substrings (146,314 strings in total). This specific substring length was chosen because at this length, every 141-character long substring of the original sequence is unique. Furthermore it was the shortest substring length for which this property held: substring lengths of 140 characters or shorter always produced duplicate substrings. As the input to a general suffix tree and the greedy SCS algorithm is a *set* of strings, these duplicate substrings would be eliminated. This would in turn make it impossible for the algorithm to correctly reassemble the original sequence: it would have no way of knowing that a particular substring had appeared multiple times in the original sequence. Note that this is not an unrealistic substring length to choose: 141-characters is within the bounds of current high volume "short-read" sequencing technologies which produce read lengths of between 36 and 150 characters.

The massive increase in dataset size (concatenating the set of 141-character substrings for input into the general suffix tree produced a 20,776,588-character string), meant that there was a large increase in main memory requirements. The program's need to access more than 2 GB of memory required it to be compiled in 64-bit rather than 32-bit. This was achieved in Visual Studio using the Configuration Manager: the default target environment is 32 bit so I created a new configuration which targets the x64 architecture. The x64 architecture is the current 64-bit standard supported by both Intel and AMD.

To maximize the available memory, I used an Amazon instance with the largest possible RAM. Amazon calls this the "High-Memory Quadruple Extra Large Instance" and it has the following specifications:

68.4 GB of memory
26 EC2 Compute Units (8 virtual cores with 3.25 EC2 Compute Units each)
64-bit platform
I/O Performance: High

Once the instance was setup and attached to my previously created EBS volume containing the datasets it was simply a case of copying the compiled 64-bit program to the instance and running it.

The program completed in 41 minutes and 49 seconds, with the various phases of the program breaking down as follows:

Phase	**Time taken**	**Peak memory**
General suffix tree construction	28 min 41 seconds	28.0 Gbyte
Node labelling	8 min 35 seconds	29.5 Gbyte
Greedy SCS algorithm	4 min 33 seconds	29.5 Gbyte

Figure 6.2: *Table showing running times and peak memory requirements for the various phases of the Swinepox virus reconstruction.*

The constructed string was slightly longer than the original DNA sequence, with a length of 146,594 characters versus 146,454 in the original. However, the two strings were extremely similar, with the constructed string consisting of two large regions that were identical to the original separated by small amounts of 'junk' sequence. The constructed string can be found in the submitted file "`Swinepox_Greedy_SCS.txt`".

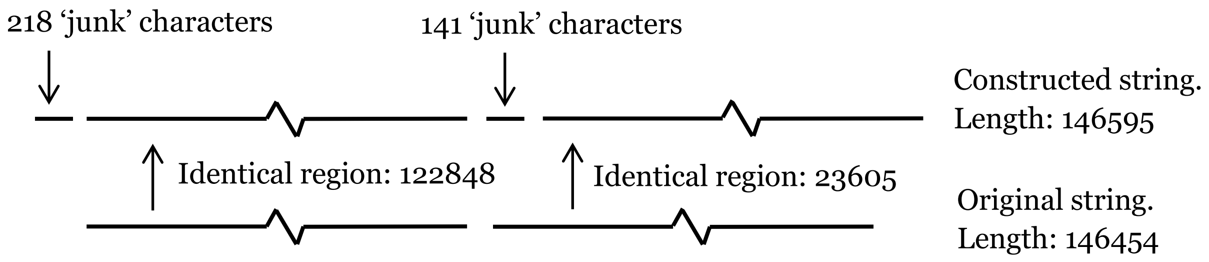


Figure 6.3: *Comparing the original and reconstructed Swinepox sequence*

To understand how the greedy algorithm could be misled (even with 'perfect' information available to it) consider that when presented with a number of equally large overlaps, the algorithm does not 'intelligently' decide which two overlapping strings to merge: it simply picks the first one it comes across.

For example, consider the following string and set of substrings:

Original sequence:	A	G	T	C	G	T	C	A	A
1	A	G	T	C					
2		G	T	C	G				
3			T	C	G	T			
4				C	G	T	C		
5					G	T	C	A	
6						T	C	A	A

Figure 6.4: *An optimal assembly for a set of 4-character strings*

Ideally the greedy algorithm would merge S_6, S_5, S_4, S_3, S_2 and then S_1 returning the original sequence "AGTCGTCAA".

However, note that there is overlap of three characters between S_5 and S_1. This is an equally valid overlap for the algorithm to pick as a second merge, but the resulting superstring is suboptimal:

Assembled string:	G	T	C	G	T	C	A	G	T	C	A	A
1							A	G	T	C		
5								G	T	C	A	
6									T	C	A	A
2	G	T	C	G								
3		T	C	G	T							
4			C	G	T	C						

Figure 6.5: *A suboptimal assembly for a set of 4-character strings.*

6.4 Comparing the naïve approach

At this stage, an enormous amount of effort had been expended on research, design and implementation. Almost one thousand lines of code had been written. Naturally, I wished to check whether this effort had been worthwhile by comparing the run-time of the suffix tree approach with a naïve implementation using the inbuilt string processing functions of C++.

A naïve algorithm to detect overlaps between two strings in C++ could use the C++ `substr()` function to extract substrings and test for potential matches. For example, when comparing the strings "*awesome*" and *"someday"*, the algorithm would first compare "*awesome*" and "*someday*". Finding that these substrings do not match, the algorithm would then test a progressively smaller suffix/prefix of the left/right string, for example:

```
"wesome" = "someda"   ? No match
"esome" = "somed"     ? No match
"some" = "some"       ? Match! Return length of overlap (4)
```

Writing C++ code for this approach took only a few minutes, and relatively few lines of code:

```
int overlap(string left, string right)
{
	for (int i = 0; i < left.length(); i++)
	{
		if (left.substr(i) == right.substr(0, left.length() - i))
			return left.length() - i;
	}
	return 0;
}
```

Algorithm 6.1: *Naïve C++ function for overlap detection.*

It's worth nothing that this is in fact *less* complete than the suffix tree approach: the search is abandoned when the largest overlap is found whereas the suffix tree approach finds *all* of the possible suffix-prefix overlaps between two strings.

To make a fair comparison with the suffix tree approach I again used the "High-Memory Quadruple Extra Large Instance" available on Amazon's cloud. I ran the overlap algorithm 100,000 times on a random sample of pairs from the set of 141-character strings extracted from the Swinepox sequence in Section 6.3.

The results are summarized in the following table:

Phase	**Time taken**	**Peak memory requirement**
Naïve overlap of 100k pairs of strings	40 seconds	< 1 GB

Figure 6.6: *Time performance of the naïve C++ approach*

Recalling that the extracted set contained 146,314 unique 141-character strings we can now extrapolate the running time of the naïve approach as follows:

Each one of the 146,314 strings must be compared to the other 146,313, therefore the `overlap()` function is called a total of (`146,314 × 146,313 = 21,407,640,282`) times.

To compute the total time taken to resolve the all-pairs suffix-prefix problem:

```
(21,407,640,282 ÷ 100,000) × 40 seconds = 8,563,056 seconds
```

This is approximately 99 days; the suffix tree approach, which took only 37 minutes to construct and label the tree for the same set of strings, compares favourably.

6.5 Computational complexity of the suffix tree algorithm

Given that the theoretical running time and space complexity of the suffix tree algorithm is `O(n)`, I wished to verify that my implementation exhibited similar behaviour. To confirm this, I tested the suffix tree construction on strings of 1 million to 5 million characters in length, again using a high performance Amazon computer. The results follow:

Number of characters	Time	Memory
1,000,000	21 seconds	1.1 GB
2,000,000	45 seconds	2.3 GB
3,000,000	72 seconds	3.4 GB
4,000,000	100 seconds	4.5 GB
5,000,000	129 seconds	5.7 GB

Figure 6.7: *Running time and memory requirements for suffix trees between 1 million and 5 million characters long.*

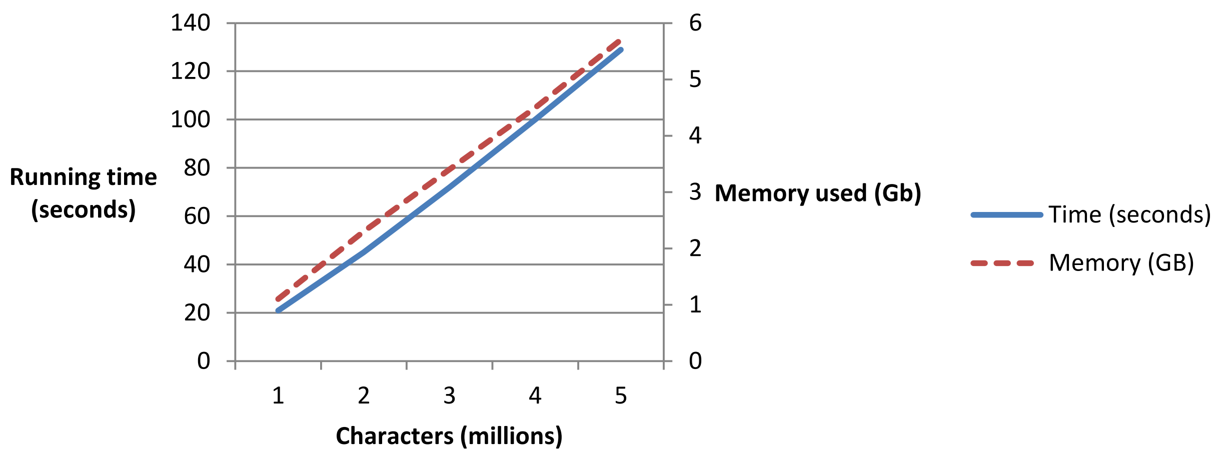


Figure 6.8: *Graphic representation of data in Figure 6.7.*

The running time and memory requirements are basically in-line with expectations. However there is a slight degradation in running time as the size of the dataset increases. A running time of 45 seconds for 2 million characters would suggest a running time of 90 seconds for 4 million characters; the actual running time for 4 million characters was 100 seconds. Further investigation would be required to determine why this is.

7. Evaluation

7.1 Evaluation of code

There are some aspects of the submitted program code that, with more development time, could be improved:

7.1.1 Use of object-oriented principles

Although the code is organised into classes, encapsulation is not properly used: in fact all member functions and variables are public. The suffix tree class itself does have general uses outside this program and could benefit from a properly implemented class interface.

7.1.2 Rule of three

As the Suffix Tree structure makes extensive use of pointers, the default C++ destructor, copy constructor and copy assignment operator would not function as expected for a user of the suffix tree class. This is commonly referred to as the "Rule of 3" in C++ programming: that is, if a class defines any one of the mentioned operators, it should probably define all three.

7.2 Running the program on larger datasets

Although this project made a partially successful attempt to recreate a Swinepox genome (with many simplifying assumptions), it was unsuccessful when faced with larger genomes.

I was kindly provided with a reference sequence for 'Mycobacterium tuberculosis' (MTB) by Dr. Taane Clark and Francesc Coll at the London School of Hygiene and Tropical Medicine. This reference sequence had a length of 4.4 million characters, considerably longer than the 146k character Swinepox sequence.

After creating some simulated reads in a process similar to that described in Section 6.3, it became clear that the program in its current state would not manage to assemble the TB virus on a 64GB RAM machine.

Even using shorter read lengths to try and reduce the size of the input dataset was not sufficient to compute an assembly. There are over 4.3 million unique 36-character substrings in the MTB sequence. Concatenating these substrings with terminal symbols (ie. in preparation to construct a general suffix tree) would result in an input string of over 159-million characters. Recalling from Section 7.3 that the algorithm used 28 GB to construct the general suffix tree for 20 million characters, we can estimate the main memory requirements of a 159 million character suffix tree to be approximately 222 GB.

In retrospect, there are several designs decisions that might have been revisited given the sheer scale of the 'real life' datasets a program such as this must deal with:

7.2.1 Speed vs. Space trade-offs

During the implementation, I consistently made decisions in favour of speed for large datasets over space efficiency. An example of this was the use of the C++ `map` container to represent children of nodes, as per Section 4.2.3. Although there were some nodes with many children which slowed the algorithm down, the average number of children per node was just less than three.

An alternative to the `map` is a simple `vector` of nodes. I conducted a small experiment comparing the use of a `vector` with the map approach on a general suffix tree for a set of 146,306 36-character strings. The result follows:

Approach	**Time taken**	**Peak memory requirements**
`map`	272 seconds	5.5 GB
`vector`	1206 seconds	3.8 GB

Figure 7.1: *Comparing the running time and memory requirements of the general suffix tree algorithm using a* `vector`*-based versus* `map`*-based approach for representing children of nodes.*

It is important to note that the `vector` approach was extremely naïve with no optimizations applied at all. We could keep the `vector` sorted, extract children using binary search, use lazy deletion when removing children, reserve capacity more intelligently or perhaps combine the use of `map` and `vector` (using the `map` only when the number of children exceeds a certain count). It seems likely that these optimizations could yield a competitive running time, and it is certainly worth investigating further given the high memory cost of the `map` approach.

7.2.2 Alternative data structures

There are a number of variations on the standard suffix tree approach that could yield more space efficiencies. These include suffix arrays [14] and various forms of compression for suffix trees [16]. In some cases, space efficiencies are traded for a slight degradation in theoretical performance which in practice is not impactful.

If the project was to be extended, it would be fruitful to investigate these alternative data structures and compare them with the approach taken here.

7.2.3 Disk based approaches

There are various disk-based algorithms for building suffix trees that overcome the problem of constrained RAM, for example [15][18]. Although Ukkonen's algorithm could theoretically be used to build a suffix tree on a disk, it would be extremely slow. Ukkonen's algorithm and suffix trees in general exhibit very poor locality. Consider that suffix links allow the algorithm to move rapidly around disparate sections of the tree structure.

As moving sections of the tree back and forth from the hard disk is very costly, the disk-based approaches aim to reduce this by intelligently partitioning the suffix tree structure into segments that maximize locality (therefore minimizing disk I/O).

Appendix A: Running and Installing the Software

The program has been compiled and tested using Microsoft's Visual Studio C++. The `main` program code contains a short example demonstrating the construction and logging of a small suffix tree. It also executes a test function in `TestSuite.h` which contains further code examples, particularly of reading string sets in the FASTA format and generating superstrings using the greedy algorithm (see function "`GREEDY_SCS_TEST`").

Sets of strings are input into the program using the FASTA file format. Each string is preceded by a single line starting with a '>' character, however, strings may span multiple lines. For example, this is a valid two-string input file to the program:

```
> string 1
ababababababab
babababababaab
> string 2
dbdbdbdbdbdbdb
```

Appendix B: Program code

B.1 main.cpp

```
#include "TestSuite.h"
#include "SuffixTree.h"

using namespace std;

int main() {
	SuffixTree tree;
	tree.construct("xabxa$");
	tree.log_tree();

	EXECUTE_TEST_SUITE();
}
```

B.2 TestSuite.h

```
#pragma once

#include <iostream>
#include <string>
#include <vector>
#include <set>

#include "SuffixTree.h"
#include "FASTA_FileReader.h"
#include "GeneralSuffixTree.h"
#include "Assembler.h"

bool SUFFIX_TREE_TEST();
bool GREEDY_SCS_TEST();
bool FASTA_FILE_READER_TEST();

void EXECUTE_TEST_SUITE() {
	std::cout << "Running tests..." << std::endl;
	typedef bool (*Test)();
	std::vector<Test> tests;
	tests.push_back(SUFFIX_TREE_TEST);
	tests.push_back(GREEDY_SCS_TEST);
	tests.push_back(FASTA_FILE_READER_TEST);

	for (int i = 0; i < tests.size(); i++) {
		std::cout	<< "Test " << i + 1
					<< (tests[i]() ? " PASSED" : " FAILED")
					<< std::endl;
	}
	std::cout << "Press any key to exit..." << std::endl;
	std::cin.get();
}
```

```
bool SUFFIX_TREE_TEST() {
       FASTA_FileReader file("Swinepox_NC_003389_complete.fasta");
       std::set<std::string> sequences = file.parse();
       SuffixTree st;
       st.construct(*sequences.begin() + "$");
       std::string test = "TGTAACCT";
       std::vector<int> v = st.get_exact_matches(test);

       if (v.size() == 3
              && v[0] == 146447
              && v[1] == 138
              && v[2] == 36082
              && sequences.size() == 1)
              return true;
       else return false;
}

bool GREEDY_SCS_TEST() {
       FASTA_FileReader file("Greedy_SCS_TestSet.fasta");
       GeneralSuffixTree gst(file.parse());
       Assembler assembler;
       assembler.label_nodes(gst);
       assembler.greedy_SCS(gst);
       if (assembler.get_SCS(gst) == "aabcaacababaaddd")
              return true;
       else return false;
}

bool FASTA_FILE_READER_TEST() {
       FASTA_FileReader file("FASTA_FileReader_TestFile.fasta");
       std::set<std::string> strings = file.parse();

       std::set<std::string>::iterator it = strings.begin();

       if (strings.size() != 5
              || (*it++).length() != 88023
              || (*it++).length() != 419
              || (*it++).length() != 34536
              || (*it++).length() != 276
              || (*it).length() != 19502)
                     return false;
       return true;
}
```

B.3 SuffixTree class

```
#pragma once

#include <string>
#include <vector>

class Suffix;
class Node;
class SuffixTree
{
public:
    SuffixTree();
    void construct(std::string);
    void log_tree();
    void log_node(Node*);
    std::vector<int> get_exact_matches(std::string) const;
    std::vector<int> retrieve_leaves(const Suffix&) const;
    std::string get_substr(int, int);
    Suffix match_string(std::string) const;
    enum Rule {RULE_2, RULE_3};          //Suffix Extension rules (Gusfield, 1997)
    void SPA(int);                //SPA: Single Phase Algorithm (Gusfield, 1997)
    Rule SEA(Suffix&, int, int);  //SEA: Single Extension Algorithm (Gusfield, 1997)
    Suffix get_suffix(Node*, int, int); //The 'skip/count' traversal method
    void RULE2(Suffix&, int, int);       //Suffix Extension Rule 2 (Gusfield, 1997)

    std::string tree_string;
    Node* root;
    int internal_node_ID;
    int length;
    int* current_end;
    Node* last_leaf_extension;
};
```

```
#include "SuffixTree.h"
#include "Node.h"
#include "Suffix.h"
#include <map>
#include <iostream>

SuffixTree::SuffixTree() {
    internal_node_ID = 0;
    current_end = new int(0);
    root = new Node(NULL, 1, new int (0), internal_node_ID);
    root->suffix_link = root;
}

void SuffixTree::construct(std::string s) {
    length = s.length();
    tree_string = '$' + s;

    (*current_end)++;
    last_leaf_extension = new Node(root, 1, current_end, 1);
    root->add_child(*this, last_leaf_extension);

    for (int i = 1; i < length; i++)
        SPA(i);
}
```

```cpp
//SPA: Single Phase Algorithm (Gusfield, 1997)
void SuffixTree::SPA(int i) {
    Suffix previous_suffix(last_leaf_extension, *current_end);
    (*current_end)++;

    int j;
    for (j = (last_leaf_extension->ID + 1); j <= (i + 1); j++) {
       Rule rule_applied = SEA(previous_suffix, j, i);
              if (rule_applied == RULE_3)
                     break;
    }
}

//SEA: Single Extension Algorithm (Gusfield, 1997)
SuffixTree::Rule SuffixTree::SEA(Suffix& previous_suffix, int j, int i) {
       int begin_index, end_index;
       Node* origin = previous_suffix.walk_up(begin_index, end_index);
       Suffix suffix = (origin == root ? get_suffix(root, j, i)
              : get_suffix(origin->suffix_link, begin_index, end_index));

       Rule rule_applied;
       if (suffix.RULE2_conditions(*this, i + 1)) {
              RULE2(suffix, i + 1, j);
              rule_applied = RULE_2;
       }
       else rule_applied = RULE_3;

       if (previous_suffix.new_internal_node)
              previous_suffix.node->suffix_link = suffix.node;
       previous_suffix = suffix;
       return rule_applied;
}


//The 'skip/count trick' for suffix tree traversal (Gusfield, 1997)
Suffix SuffixTree::get_suffix(Node* origin, int begin_index, int end_index) {
       int char_index = *origin->end_index;

       while (begin_index <= end_index) {
              origin = origin->get_child(*this, begin_index);
              if (origin->edge_length() < end_index - begin_index + 1)
                     char_index = *origin->end_index;
              else char_index = origin->begin_index + (end_index - begin_index);
              begin_index+=origin->edge_length();
       }
       return Suffix(origin, char_index);
}
```

```
//Match a string from the root of the tree
Suffix SuffixTree::match_string(std::string string) const {
    int char_index;
    Node* current_node = root;
    while (!string.empty()) {
       current_node = current_node->get_char_child(*this, string[0]);
       if (current_node == NULL)
              return Suffix(NULL, 0);
       else {
              char_index = current_node->begin_index;
              int i = 1;
              for (; i < string.length() && i < current_node->edge_length(); i++)
                     if (string[i] != tree_string[char_index + i])
                            return Suffix(NULL, 0);
              string.erase(0, i);
       }
    }
    return Suffix(current_node, char_index);
}

std::vector<int> SuffixTree::get_exact_matches(std::string string) const {
       Suffix suffix = match_string(string);
       if (suffix.node == NULL)
              return std::vector<int>();
       else
              return retrieve_leaves(suffix);
}

//depth first tree traversal to gather leaf IDs below a given suffix
std::vector<int> SuffixTree::retrieve_leaves(const Suffix& suffix) const {
       std::vector<int> leaf_IDs;
       std::vector<Node*> nodes_to_visit (1, suffix.node);

       while (!nodes_to_visit.empty()) {
              Node* current_node = nodes_to_visit.back();
              nodes_to_visit.pop_back();
              if (current_node->is_leaf())
                     leaf_IDs.push_back(current_node->ID);
              else
                     current_node->get_children(nodes_to_visit);
       }
       return leaf_IDs;
}


std::string SuffixTree::get_substr(int start_pos, int end_pos) {
    if (start_pos > end_pos)
       return std::string();
    return tree_string.substr(start_pos, end_pos - start_pos + 1);
}

void SuffixTree::RULE2(Suffix& suffix, int char_index, int new_leaf_ID) {
    if (!suffix.ends_at_node()) {  //eg. case 2 (path ends inside an edge)
       suffix.node->split_edge(*this, suffix.char_index, --internal_node_ID);
       suffix.node = suffix.node->parent;
       suffix.new_internal_node = true;
    }
    Node* new_leaf = new Node(suffix.node, char_index, current_end, new_leaf_ID);
    suffix.node->add_child(*this, new_leaf);
    last_leaf_extension = new_leaf;
}
```

```
void SuffixTree::log_tree() {
       freopen("tree.gv", "w", stdout);
       std::cout << "digraph g {" << std::endl;
       log_node(root);
       std::cout << "}" << std::endl;
       freopen( "CON", "w", stdout );
}

void SuffixTree::log_node(Node* parent) {
       int parent_ID = parent->ID;
       std::map<int, Node*>::iterator it = parent->children.begin();
       for (; it != parent->children.end(); it++) {
           Node* current_child = it->second;
           std::cout << "\"" << parent->ID << "\" -> " << "\""
               << current_child->ID << "\"" << " [label = \""
               << get_substr(current_child->begin_index, *current_child->end_index)
               << "\"];" << std::endl;
           log_node(current_child);
        }
}
```

B.4 Suffix class

```
#pragma once

class SuffixTree;
class Node;
class Suffix {
public:
    Suffix(Node*, int);
    bool ends_at_node() const;
    bool ends_at_leaf() const;
    bool continues_with_char(const SuffixTree&, int) const;
    bool RULE2_conditions(const SuffixTree&, int) const;
    bool new_internal_node;
    Node* walk_up(int&, int&) const;

    Node* node;
    int char_index;
};
```

```
#include "Suffix.h"
#include "Node.h"
#include "SuffixTree.h"

Suffix::Suffix(Node* n, int c) : node(n), char_index(c) {
    new_internal_node = false;
}

bool Suffix::ends_at_node() const {
    return char_index == *node->end_index;
}

bool Suffix::ends_at_leaf() const {
    return node->is_leaf() && ends_at_node();
}
```

```
bool Suffix::continues_with_char(const SuffixTree& tree, int tree_index) const {
    char ch = tree.tree_string[tree_index];
    bool terminal(ch == '$');
    return (ends_at_node() && node->get_child(tree, tree_index) != NULL)
            || (!ends_at_node() && tree.tree_string[char_index + 1] == ch
                           && (!terminal || char_index + 1 == tree_index));
}

Node* Suffix::walk_up(int& begin_index, int& end_index) const {
       if (ends_at_node() && node->suffix_link != NULL) {
              begin_index = *node->end_index;
              end_index = *node->end_index - 1;
              return node;
       }
       else {
              begin_index = node->begin_index;
              end_index = char_index;
              return node->parent;
       }
}

bool Suffix::RULE2_conditions(const SuffixTree& tree, int tree_index) const {
       return !ends_at_leaf() && !continues_with_char(tree, tree_index);
}
```

B.5 Node class

```
#pragma once

#include <string>
#include <vector>
#include <map>

class SuffixTree;
class Node
{
public:
       Node(Node*, int, int*, int);
       int edge_length() {return *end_index - begin_index + 1;}
       void add_child(const SuffixTree&, Node*);
       void remove_child(const SuffixTree&, Node*);
       bool is_leaf() {return children.empty();}
       void split_edge(const SuffixTree&, int, int);
       Node* get_child(const SuffixTree&, int char_index);
       Node* get_char_child(const SuffixTree&, char ch);
       void get_children(std::vector<Node*>&) const;
       int get_key(const SuffixTree&, Node*, int) const;

       Node* parent;
       std::map<int, Node*> children;
       std::vector<int> labels;
       Node* suffix_link;
       int begin_index;
       int* end_index;
       int ID;
};
```

```cpp
#include "Node.h"
#include "SuffixTree.h"

Node::Node(Node* parent, int begin_index, int* end_index, int ID) {
       this->parent = parent;
       this->begin_index = begin_index;
       this->end_index = end_index;
       this->ID = ID;
       suffix_link = NULL;
}

void Node::add_child(const SuffixTree& tree, Node* child_to_add) {
       int key = get_key(tree, child_to_add, child_to_add->begin_index);
       children[key] = child_to_add;
}

void Node::remove_child(const SuffixTree& tree, Node* child_to_remove) {
       int key = get_key(tree, child_to_remove, child_to_remove->begin_index);
       children.erase(key);
}

int Node::get_key(const SuffixTree& tree, Node* node, int index) const {
       char ch = tree.tree_string[index];
       return (ch != '$' ? ch * (-1) : index);
}


void Node::split_edge(const SuffixTree& tree, int char_index, int new_node_ID) {
    Node* new_node = new Node(parent, begin_index, new int(char_index), new_node_ID);
    parent->remove_child(tree, this);
    parent->add_child(tree, new_node);

    this->parent = new_node;
    this->begin_index = char_index + 1;
    new_node->add_child(tree, this);
}

Node* Node::get_child(const SuffixTree& tree, int char_index)
{
    int key = get_key(tree, this, char_index);
    std::map<int, Node*>::iterator it = children.find(key);
    if (it != children.end())
       return it->second;
    else
       return NULL;
}

void Node::get_children(std::vector<Node*>& ret_children) const {
       std::map<int, Node*>::const_iterator it = children.begin();
       for (; it != children.end(); it++)
              ret_children.push_back(it->second);
}

Node* Node::get_char_child(const SuffixTree& tree, char ch) {
       std::map<int, Node*>::iterator it = children.find(ch * (-1));
       if (it != children.end())
              return it->second;
       else
              return NULL;
}
```

B.6 GeneralSuffixTree class

```
#pragma once

#include <set>
#include <string>
#include <map>
#include "SuffixTree.h"
#include "Overlap.h"
#include "StringMap.h"

class GeneralSuffixTree : public SuffixTree
{
public:
	GeneralSuffixTree(std::set<std::string>);
	typedef std::map<int, StringMap>& Mapping;
	Overlap lookup(Overlap, Mapping) const;
	std::string extract_string(int) const;

};


#include "GeneralSuffixTree.h"
#include "Node.h"

GeneralSuffixTree::GeneralSuffixTree(std::set<std::string> strings)
{
	std::string concat;
	std::set<std::string>::iterator it;
	for (it = strings.begin(); it != strings.end(); it++)
		concat += (*it + "$");
	SuffixTree::construct(concat);
}

Overlap GeneralSuffixTree::lookup(Overlap to_lookup, Mapping strings) const
{
	int depth = to_lookup.overlap;
	int string_right = to_lookup.string_right;
	int overlap = to_lookup.overlap;
	Node* node = to_lookup.node;
	while(true) {
		for (int i = 0; i < node->labels.size(); i++) {
			int string_left = node->labels[i];
			if (string_left != string_right
			    && string_left != (strings[string_right]).right_end
			    && !strings[string_left].deleted)
			       return Overlap(node, string_left, string_right, depth);
		}
		depth -= node->edge_length();
		node = node->parent;
	}
}

std::string GeneralSuffixTree::extract_string(int string_ID) const {
	std::string to_return;
	for (int i = string_ID - 1; tree_string[i] != '$'; i--)
		to_return = tree_string[i] + to_return;
	return to_return;
}
```

B.7 FASTA_FileReader class

```
#pragma once

#include <fstream>
#include <string>
#include <set>

class FASTA_FileReader {
public:
    FASTA_FileReader(std::string);
    std::set<std::string> parse();
    std::ifstream infile;
};


#include "FASTA_FileReader.h"
#include <cassert>

FASTA_FileReader::FASTA_FileReader(std::string filename) {
    infile.open(filename.c_str());
       assert(!infile.fail());
}

std::set<std::string> FASTA_FileReader::parse() {
    std::set<std::string> to_return;
    std::string current_string;
    std::string current_line;
    assert(getline(infile, current_line) && current_line[0] == '>');

    while (std::getline(infile, current_line)) {
       if (!current_line.empty() && current_line[0] != '>')
              current_string+=current_line;
       else if (!current_string.empty()) {
              to_return.insert(current_string);
              current_string.clear();
       }
    }
    if (!current_string.empty())
       to_return.insert(current_string);
    return to_return;
}
```

B.8 Assembler class

```
#pragma once
#include <string>
#include <vector>
#include <map>

class StringMap;
class Overlap;
class GeneralSuffixTree;
class Assembler
{
public:
	Assembler();
	void label_nodes(GeneralSuffixTree&);
	void push_overlap(GeneralSuffixTree& gst, std::string, int);
	void initialise_greedy_SCS(GeneralSuffixTree&);
	void greedy_SCS(GeneralSuffixTree&);
	int merge_strings(Overlap);
	std::string get_SCS(GeneralSuffixTree&);

	std::vector<Overlap> overlaps;
	std::map<int, StringMap> mapping;
	int left_handle;
};


#include <algorithm>
#include "Assembler.h"
#include "GeneralSuffixTree.h"
#include "Node.h"
#include "CompareOverlap.h"

Assembler::Assembler()
{
	left_handle = -1;
}

void Assembler::label_nodes(GeneralSuffixTree& gst) {
	std::vector<Node*> to_visit (1, gst.root);
	while (!to_visit.empty()) {
		Node* current_node = to_visit.back();
		to_visit.pop_back();

		std::vector<Node*> children;
		current_node->get_children(children);
		for (int i = 0; i < children.size(); i++) {
			if (gst.tree_string[children[i]->begin_index] == '$')
				current_node->labels.push_back(children[i]->begin_index);
		}
		to_visit.insert(to_visit.end(), children.begin(), children.end());
	}
}
```

```
void Assembler::initialise_greedy_SCS(GeneralSuffixTree& gst) {
        std::string to_match;
        for (int i = 1; i < gst.tree_string.length(); i++) {
                if (gst.tree_string[i] != '$')
                        to_match += gst.tree_string[i];
                else {
                        mapping.insert(std::pair<int, StringMap>(i, StringMap(i)));
                        push_overlap(gst, to_match, i);
                        to_match.clear();
                }
        }
}

void Assembler::push_overlap(       GeneralSuffixTree& gst,
                                    std::string string, int string_right) {
        int depth = 0;
        int deepest_overlap;
        Node* node = gst.root;
        Node* deepest_node;
        while (!string.empty()) {
                if (!node->labels.empty()) {
                        deepest_overlap = depth;
                        deepest_node = node;
                }
                node = node->get_char_child(gst, string[0]);
                depth += node->edge_length();
                string.erase(0, node->edge_length());
        }
        overlaps.push_back(Overlap( deepest_node,
                                    deepest_node->labels.front(),
                                    string_right,
                                    deepest_overlap));
}


void Assembler::greedy_SCS(GeneralSuffixTree& gst) {
        initialise_greedy_SCS(gst);
        std::make_heap (overlaps.begin(), overlaps.end(), CompareOverlap());
        while (overlaps.size() > 1) {
                std::pop_heap(overlaps.begin(), overlaps.end(), CompareOverlap());
                Overlap current_overlap = overlaps.back();
                overlaps.pop_back();
                Overlap lookup_overlap = gst.lookup(current_overlap, mapping);

                if (lookup_overlap.overlap == current_overlap.overlap)
                        left_handle = merge_strings(lookup_overlap);
                else {
                        overlaps.push_back(lookup_overlap);
                        push_heap (overlaps.begin(), overlaps.end(), CompareOverlap());
                }
        }
}
```

```
int Assembler::merge_strings(Overlap overlap)
{
	int string_left = overlap.string_left;
	int string_right = overlap.string_right;
	int string_left_leftend = mapping[string_left].left_end;
	int string_right_rightend = mapping[string_right].right_end;
	mapping[string_right].left = string_left;
	mapping[string_left].right = string_right;
	mapping[string_left_leftend].right_end = mapping[string_right].right_end;
	mapping[string_right_rightend].left_end = mapping[string_left].left_end;
	mapping[string_left].deleted = true;
	mapping[string_left].suffix_overlap = overlap.overlap;
	return string_left_leftend;
}

std::string Assembler::get_SCS(GeneralSuffixTree& gst)
{
	int current_string_ID = left_handle;
	std::string to_return = gst.extract_string(current_string_ID);
	while (mapping[current_string_ID].right != -1)
	{
	    int overlap = mapping[current_string_ID].suffix_overlap;
	    current_string_ID = mapping[current_string_ID].right;
	    std::string to_add = gst.extract_string(current_string_ID).substr(overlap);
	    to_return += to_add;
	}
	return to_return;
}
```

B.9 Overlap class

```
#pragma once
class Node;
class Overlap
{
public:
	Overlap(Node*, int, int, int);

	Node* node;
	int string_left;
	int string_right;
	int overlap;
};
#include "Overlap.h"

Overlap::Overlap(Node* n, int left, int right, int overlap)
{
	node = n;
	string_left = left;
	string_right = right;
	this->overlap = overlap;
}
```

B.10 CompareOverlap class

```
#pragma once

#include "Overlap.h"
class CompareOverlap
{
public:
	bool operator ()(const Overlap& left, const Overlap& right) const {
		return left.overlap < right.overlap;
	}
};
```

B.11 StringMap class

```
#pragma once
class StringMap
{
public:
	StringMap() {}
	StringMap(int);

	int left;
	int right;
	int left_end;
	int right_end;
	int suffix_overlap;
	bool deleted;
};


#include "StringMap.h"

StringMap::StringMap(int string_ID)
{
	left = right = suffix_overlap = -1;
	left_end = right_end = string_ID;
	deleted = false;
}
```

Appendix C: Bibliography

[1] Dan Gusfield (1997) Algorithms on Strings, Trees, and Sequences. Cambridge University Press.

[2] E. Ukkonen (1995) On-line construction of suffix trees. ALGORITHMICA Volume 14, Number 3 (1995), 249-260, DOI: 10.1007/BF01206331

[3] Jonathan S. Turner (1989) Approximation algorithms for the shortest common superstring problem. Information and Computation, Volume 83, Issue 1, Pages 1–20, http://dx.doi.org/10.1016/0890-5401(89)90044-8

[4] Hans-Joachim Bokenhauer, Dirk Bongartz (2007) Algorithmic Aspects of Bioinformatics. Springer.

[5] J .Gallant, D. Maier and J .A.Storer, On Finding Minimal Length Superstrings, Journal oj Computer and System Sciences, 20, 50-58, 1980.

[6] Mark Nelson (1996) Fast String Searching With Suffix Trees. Dr. Dobb's Journal (August 1996). http://marknelson.us/1996/08/01/suffix-trees/

[7] Dotan Tsadok (2002) ANSI C implementation of a Suffix Tree, Haifa University (unpublished). http://mila.cs.technion.ac.il/~yona/suffix_tree/

[8] Weiner, P. Linear pattern matching algorithms. Conf. Record, IEEE 14th Annual Symposium on Switching and Automata Theory, pp. 1-11.

[9] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. Journal of the ACM, 23(2):262-272, 1976.

[10] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Algorithmica, Volume 19, Number 3 (1997), 331-353, DOI: 10.1007/PL00009177.

[11] National Center for Biotechnology Information. Swinepox virus, complete genome in FASTA format. http://www.ncbi.nlm.nih.gov/nuccore/18640086?report=fasta

[12] American National Standards Institute. INCITS/ISO/IEC 14882-2012. Information Technology/Programming Languages/C++.

[13] William Cook. The Travelling Salesman Problem, Georgia Institute of Technology. http://www.tsp.gatech.edu/

[14] Giovanni Manzini and Paolo Ferragina. Engineering a Lightweight Suffix Array Construction Algorithm. Volume 40, Number 1 (2004), 33-50, DOI: 10.1007/s00453-004-1094-1.

[15] Benjarath Phoophakdee , Mohammed J. Zaki Genome-scale disk-based suffix tree indexing. SIGMOD '07 Proceedings of the 2007 ACM SIGMOD international conference on Management of data. Pages 833 – 844.

[16] R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees, with Applications to Text Indexing and String Matching, SIAM Journal on Computing, 35(2), 2005, 378-407.

[17] Graphviz – Graph Visualization Software. http://www.graphviz.org/Documentation.php

[18] Hunt, E, Atkinson, MP, and Irving, RW (2002) Database indexing for large DNA and protein sequence collections. Vldb Journal, 11 . pp. 256-271. (doi:10.1007/s007780200064).

[19] George Reese (2009) Cloud Application Architectures: Building Applications and Infrastructure in the Cloud: Transactional Systems for EC2 and Beyond.