Searching for unique strings of variable length in DNA using pruned and truncated suffix trees

Klaske van Vuurden

June 20, 2007

Contents

1 Introduction **3**

2 Background **5**
2.1 DNA 5
2.2 Primers 6
2.2.1 Melting temperature 6
2.2.2 Length 7
2.2.3 GC percentage 7

3 Formalisation **8**

4 Suffix tree **10**
4.1 STrie 10
4.1.1 Complexity 11
4.2 STree 11
4.2.1 Complexity 12
4.3 Online construction of $STree(T)$ 12

5 Modifications on the suffix tree **15**
5.1 A suffix tree over multiple strings 16
5.2 A truncated suffix tree 16
5.2.1 Algorithm 17
5.3 Truncated Ukkonen over multiple strings 18
5.3.1 Algorithm 18
5.3.2 Complexity 20
5.4 Pruning a tree on a non-decreasing function 23
5.4.1 Algorithm 23
5.4.2 Correctness 24
5.4.3 Complexity 25
5.4.4 Remarks 26

6 Counting substrings in suffix trees **27**
6.1 In $STree(T)$ 27
6.2 In $STree(\mathcal{T})$ 28

6.3 In $STrunc(\mathcal{T}, \Omega)$ . . . 28
6.4 In $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ . . . 28

7 Transforming back to a string 29
7.1 Notation . . . 29
7.2 Complexity . . . 30

8 Counting DNA-strands with suffix trees 31
8.1 Pruning on melting temperature . . . 31
8.2 Pruning on GC-percentage . . . 33
8.3 Pruning on multiple functions . . . 34

9 Implementation 35
9.1 Building the suffix tree . . . 36
9.1.1 Nodes . . . 37
9.1.2 Character string . . . 37
9.1.3 Filtering . . . 37
9.2 Reading from an output file . . . 38
9.2.1 Search . . . 38

10 Experiments 39
10.1 Input file . . . 39
10.2 Time . . . 40
10.3 Number of nodes . . . 42

11 Conclusions 45

12 Further research 46

Chapter 1

Introduction

Only as recent as April 2003, the project of sequencing the entire human genome was finished. This has led the way for researchers to focus on the exploration of genes, located somewhere in 23 pairs of chromosomes. To make efficient research possible, certain interesting regions of the genome need to be isolated.

Polymerase chain reaction (PCR) allows DNA from a selected region of a genome to be amplified, effectively "purifying" this DNA away from the remainder of the genome [AJL+02]. To start the replication process for this reaction, an appropriate initiator needs to be known for the region of interest. Such an initiator is called a *primer*.

At the request of Peter E. M. Taschner of the Leids Universitair Medisch Centrum (LUMC), Jeroen Laros attempts to find suitable primers of some fixed length. In [Lar05], he does not only look at unique strands, but he filters the set of possible primers for GC-percentage, melting temperature and mutual positions of pairs of primers. This filtering does not necessarily have to be exercised on a set of unique strands. Both GC-percentage and melting temperature can be calculated for all strands of fixed length of a DNA-strand, and whole regions of the DNA can be thereby disqualified as possible positions for primers.

When attempting to find a suitable primer in a DNA-strand without knowing its length in advance, it is no longer possible to filter the DNA-strand in advance. Strands will assume different melting temperatures and GC-percentages after a nucleotide is added to them.

Both this problem and that of gathering information about strands of different lengths can be addressed by the introduction of some modifications

on *suffix trees*. This thesis explores these modifications and discusses their suitability to solve the problem of finding primers.

We start by providing some background information on the subject of finding primers in the following chapter. The first part of the rest of the thesis will be about the data structures and algorithms describing the different extensions on suffix trees. The second part will describe how these extensions can be applied to the detection of primers. In the third part we will discuss an actual implementation, which was developed for this project, and we will report on experiments we have performed.

This project was conducted at the request of, and in cooperation with, Hendrik Jan Hoogeboom and Jeroen Laros of the Leiden Institute of Advanced Computer Science (LIACS).

Chapter 2

Background

The goal of this project is the detection of *unique* substrings of arbitrary length in a string representing a strand of DNA. A string is considered unique when it has a single occurrence on the genome. The resulting set of unique substrings will possibly contain a set of *primers*. In this section, some microbiological concepts are explained, and an outline on the use of primers will be given.

2.1 DNA

In [AJL+02] *deoxyribonucleic acid* (DNA) is described as follows. DNA is a set of strands of monomers. Each monomer in a single DNA strand – i.e., each nucleotide – consists of two parts: a sugar (deoxyribose) with a phosphate group attached to it, and a base, which may be either adenine (A), guanine (G), cytosine (C) or thymine (T). The abbreviations as mentioned here will be used from this point on instead of the full names of the bases. Each sugar is linked to the next via the phosphate group, creating a polymer chain composed of a repetitive sugar-phosphate backbone with a series of bases protruding from it. The DNA polymer is extended by adding monomers at one end. The bases protruding from the existing strand bind to bases of the strand being synthesised, according to a strict rule defined by the complementary structures of the bases: A binds to T, and C binds to G. In this way, a double-stranded structure is created, consisting of two exactly complementary sequences of A's, C's, T's, and G's.

Every monomer is added to the strand by connecting its $5'$-*phosphate* to a 3-OH group protruding from the monomer it is connecting to. The direction in which new monomers can be added is therefore called the $5'$-*to*-$3'$

direction of chain growth. When double stranded DNA is duplicated, the two complementary sequences are read in opposite order - i.e. always in the $5'$-to-$3'$ direction, as illustrated in figure 2.1.

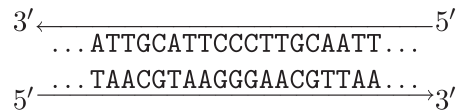

Figure 2.1: Double stranded DNA

2.2 Primers

A primer is a relatively short nucleic acid strand that serves as a starting point for DNA replication on a single DNA-strand. Here, the primer is the exact complement of the strand read from the starting position in the $3'$-to-$5'$ direction [AJL+02]. To assure that the replication can start from no other position on the strand, the primer needs to be unique. Apart from this, there are some other desired features for a primer. The ones considered in this project are *melting temperature*, length and *GC-percentage*.

2.2.1 Melting temperature

The melting temperature (Tm) of a primer is defined as the temperature at which 50% of the same DNA molecules form a stable double helix and the other 50% have been separated to single strand molecules.

Melting temperatures that are too high can cause problems since the DNA-polymerase (the enzyme that induces duplication) is less active at such temperatures. The optimal melting temperature lies between approximately 50°C and 65°C.

There are several methods available to estimate the melting temperature of a strand. The most accurate models available are based on each of the pairs of neighbouring bases on the same strand. This technique is the *nearest neighbour interaction* technique, and for this project we have used the model described by K.J. Breslauer et.al. in [BFBM86].

2.2.2 Length

The melting temperature required increases with the length of the primer. Primers that are too short would anneal at several positions on a long DNA template, which would result in non-specific copies. On the other hand, the length of a primer is limited by the temperature required to melt it. The optimal length of a primer is generally between 10 and 30 nucleotides. This has been an important motivator in the choice of the data structure chosen for this project.

2.2.3 GC percentage

The percentage of certain nucleotides within a strand can have large influence on its suitability as a primer. The more C or G nucleotides a strand contains, the higher the melting temperature will be. Furthermore, a percentage of either A and T or C and G nucleotides can cause the bases of the primer strand to form unwanted pairs among each other. A primer typically has a percentage of bases C and G around 50%.

Chapter 3

Formalisation

A DNA-strand is characterised by the sequence of bases that one of its two complementary strands is built of. Its structure can therefore be described by the string formed by the abbreviations of the bases read in $5'$-to-$3'$-direction. A DNA-strand can be described as a string $X = x_1 \dots x_L$ with $x_i \in \Sigma$, $1 \leq i \leq L$, where $\Sigma = \{A, T, C, G\}$.

Since the DNA of most living organisms contains more than one molecule i.e., *chromosome*, the DNA needs to be displayed as a set of strings $\mathcal{X} = \{X_1, \dots, X_n\}$.

Unfortunately, not every nucleotide on every position in a chromosome is known. There are certain areas that consequently cannot be described in the string representing that chromosome. Usually, unknown positions are denoted by the character N. Since we are looking for unique substrings, we need to avoid any unknown positions, and we need to treat the identified areas as separate chunks. A chromosome can be subdivided in several identified areas $T_1 \dots T_n$. The whole of an organism's DNA can then be displayed as $\mathcal{T} = \{T_{1,1} \dots T_{1,a_1} \dots T_{n,1} \dots T_{n,a_n}\}$. Now $T_{i,j} = t_{i,j,1} \dots t_{i,j,L_{i,j}}$ represents substring j of chromosome i for $1 \leq i \leq n$ and $1 \leq j \leq a_i$ with n the number of chromosomes in $\mathcal{T}$, a_i the number of identified chunks of chromosome i and $L_{i,j}$ the length of chunk $T_{i,j}$. To find the exact position of a substring on the genome, we need to keep track of the original positions of the chunks on the chromosome.

The problem of finding short unique strands of DNA in the DNA-strand can now be described as finding a substring $t_{i,j,k} \dots t_{i,j,k+l-1}$ of length l, for which there exist no i', j', k' so that $t_{i,j,k} \dots t_{i,j,k+l-1} = t_{i',j',k'} \dots t_{i',j',k'+l-1}$ and $i' \neq i$, $j' \neq j$ or $k' \neq k$.

The detection of unique substrings in a collection of strings can be achieved by different methods.

In [Lar05] Jeroen Laros describes why the specific problem of finding such substrings for a fixed length can best be approached by counting occurrences of substrings by reading through the file multiple times. He achieves a relatively low space complexity by considering each possible string of a certain length in every traversal of the file, and only counting the strings with that specific prefix.

Another widely used method for this problem is that of suffix trees. They have the added benefit that they can describe the uniqueness of strings of arbitrary length. Although they have the disadvantage that a tree representing the entire human genome will take more space than feasible with current technology, they can provide a solution for smaller sequences. We will introduce these data structures in chapter 4 together with an algorithm which creates them with linear time and space complexity as suggested by Esko Ukkonen in [Ukk95]. In chapters 5 and 6 we will introduce some modifications on these suffix trees, that will make them more suitable for the search for primers.

Chapter 4

Suffix tree

In [Ukk95] the *suffix tree* is introduced as follows. With string $T = t_1 \dots t_n$ over an alphabet Σ, a *suffix* of T is defined as $T_i = t_i \dots t_n$ and $T_{n+1} = \epsilon$ is the *empty* suffix, a suffix tree is a trie-like data structure representing the set of all suffixes of T, which is denoted by $\sigma(T)$. Each string α, such that $T = \beta\alpha\gamma$ for some strings β and γ, is a *substring* of T.

4.1 STrie

Before describing the suffix tree, the structure on which it is based, namely the suffix trie, needs to be introduced. Formally, the suffix trie representing $\sigma(T)$ is denoted as $STrie(T) = (Q \cup \{\bot\}, root, F, g, f)$ and is defined as an augmented deterministic finite-state automaton which has a tree-shaped transition graph and is augmented with the suffix function f and auxiliary state $\bot$. In this trie, Q has a one-to-one correspondence to the substrings of T. The state corresponding to substring α is denoted as $\overline{\alpha}$. Furthermore: $\overline{\epsilon} = root$, F is the set of final states, with $\overline{r} \in F \Leftrightarrow r \in \sigma(T)$.

The *transition function* g is defined as:

$$\forall\, \overline{\alpha}, \overline{\gamma} \in Q\; \forall \beta \in \Sigma : \gamma = \alpha\beta \Rightarrow g(\overline{\alpha}, \beta) = \overline{\gamma}$$

The *suffix function* f is defined as:

$$f(\overline{\alpha}) = \begin{cases} \overline{\gamma}, & \text{for } \overline{\alpha} \neq root \text{ and } \alpha = \beta\gamma \text{ for } \beta \in \Sigma \\ \bot, & \text{for } \overline{\alpha} = root \\ \text{undefined}, & \text{for } \overline{\alpha} = \bot \end{cases}$$

Now $f(r)$ is the *suffix link* of state r. In $STrie(T^i)$ with $T^i = t_1 \dots t_i$ a $prefix$ of T and F_{i-1} the final states set of $STrie(T^{i-1})$, according to the

definition of f the following holds: For any state $r \in Q$ so that $r \in F_{i-1}$ if and only if there is a j (with $0 \leq j \leq i-1$) for which $r = f^j(\overline{t_1 \dots t_{i-1}})$. The path that can be traversed through suffix links from state $\overline{t_1 \dots t_{i-1}}$ to state $\bot$ in $STrie(T)$ is the *boundary path*. Algorithm 4.1 shows how $STrie(T^i)$ is created. Here *top* denotes $\overline{t_1 \dots t_{i-1}}$. $STrie(T^i)$ is constructed from $STrie(T^{i-1})$ as follows. For every state $\overline{\alpha}$ found on a traversal of the boundary path for which $g(\overline{\alpha}, t_i)$ is undefined, add a new state $\overline{\alpha t_i}$ and update g and f with $g(\overline{\alpha}, t_i) = \overline{\alpha t_i}$ and $f(\overline{\alpha t_i}) = g(f(\overline{\alpha}), t_i)$. The traversal of the boundary path can stop as soon as a $\overline{\alpha}$ is found for which state $\overline{\alpha t_i}$ already exists. Namely, if this requirement is met, for all $\overline{\alpha'} = f^j(\overline{\alpha})$, $j \geq 1$ there must exist a state $\overline{\alpha' t_i}$ in $STrie(T^{i-1})$ and $g(\overline{\alpha'}, t_i) = \overline{\alpha' t_i}$.

4.1.1 Complexity

Algorithm 4.1: Creation of $STrie(T^i)$

1. $r \leftarrow$ top;
2. **while** $g(r, t_i)$ *is undefined* **do**
3. create new state r' and new transition $g(r, t_i) = r'$;
4. **if** $r \neq$ *top* **then** create new suffix link $f(\text{oldr}') = r'$;
5. $\text{oldr}' \leftarrow r'$;
6. $r \leftarrow f(r)$;
7. create new suffix link $f(\text{oldr}') = g(r, t_i)$;
8. top $\leftarrow g(\text{top}, t_i)$;

In the creation of $STrie(T^i)$ from $STrie(T^{i-1})$ at most i new states are created, since the boundary path has a length of at most i final states. In the creation of $STrie(T)$ from $STrie(\epsilon)$ at most $\sum_{i=1}^{|T|} i$ new states with their corresponding transition functions and suffix links are created. The worst case for both time and space complexity are therefore $\mathcal{O}(|T|^2)$.

4.2 STree

It is possible to represent $STrie(T)$ more efficiently. By only using the leaves and the nodes that have more than one child (*branching nodes*), and representing strings by their indices, it is possible to represent $STrie(T)$ in space linear to the length of T. Suffix tree $STree(T)$ is a data structure that represents $STrie(T)$ in this way.

In $STree(T) = (Q' \cup \{\bot\}, root, g', f')$, Q' consists of all leaves of $STrie(T)$

and all branching states R as defined below:

$$R = \{r \in Q | \exists a, b \in \Sigma, \exists s, t \in Q : a \neq b \ \wedge \ g(r, a) = s \wedge g(r, b) = t\}$$

States $Q' \cup \{\perp\}$ are the *explicit states*. The *implicit states* are the states of $STrie(T)$ not contained in $Q' \cup \{\perp\}$. The generalised transition function g' is defined for explicit states s and r and a string β over alphabet Σ as $g'(s, \beta) = r$ if and only if r is a child of s in $STree(T)$ and the transition path from states s to r in $STrie(T)$ spells out the string β. If $\beta = t_k \dots t_p$ we can also write $g'(s, \beta) = r$ as $g'(s, (k, p)) = r$. If $t_k = a$, $g'(s, (k, p)) = r$ is called an *a-transition*.

The suffix function becomes:

$$f'(\overline{\alpha}) = \begin{cases} \overline{\gamma}, & \text{for } \overline{\alpha} \neq root, \quad \overline{\alpha} \text{ is a branching state} \\ & \qquad\qquad\qquad \text{and } \alpha = \beta\gamma \text{ for } \beta \in \Sigma \\ \perp, & \text{for } \overline{\alpha} = root \\ \text{undefined}, & \text{for } \overline{\alpha} = \perp \end{cases}$$

If $f'(\overline{\alpha}) = \overline{\gamma}$ for $\overline{\alpha}$ a branching state, $\overline{\gamma}$ is also a branching state, since the occurrence of more than one different substring with prefix $\beta\gamma$ implies that substrings with prefix γ will also occur multiple times.

An explicit or implicit state r of the suffix tree can be referred to by a *reference pair* $(s, (k, p))$, where s is an explicit state, and the string $t_k \dots t_p$ is spelled out by the transition path in $STrie(T)$ from s to r. If s is the closest ancestor of r in $STree(T)$, then $(s, (k, p))$ is *canonical*.

4.2.1 Complexity

$STree(T)$ contains at most $|T|$ leaves, 1 for each suffix. Therefore it has at most $|T| - 1$ branching states (when $|T| > 1$) and at most $2|T| - 2$ transitions. The space needed for $STree(T)$ is at most $\mathcal{O}(|T|)$.

4.3 Online construction of $STree(T)$

There are several algorithms for the construction of suffix trees. Edward McCreight suggests a procedure for construction in linear time in [McC76]. Esko Ukkonen suggests a similar method in [Ukk95] that constructs $STree(T)$ *online*, – i.e. it only needs prefix T^i to create $STree(T^i)$ and no knowledge of suffix T_{i+1} is required. Ukkonen's algorithm for the online construction of a suffix tree is based on the same principle as algorithm 4.1. For this project

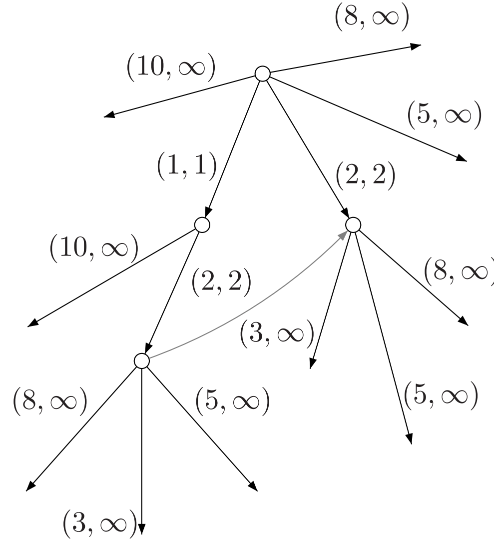

Figure 4.1: $STree$(kokoskoek$)

the Ukkonen's approach was chosen since it is online and it does not require any previous knowledge about the input string.

The set of states in $STree(T^{i-1})$ that needs to be updated to obtain $STree(T^i)$ is only a subset of the corresponding states needed to obtain $STrie(T^i)$ from $STrie(T^{i-1})$. In algorithm 4.1 all states on the boundary path of $STrie(T^{i-1})$ that do not have a t_i-transition are updated. By making a slight change in transition function g for $STree(T)$, it is no longer necessary to update the states that are leaves in $STrie(T^{i-1})$. Any transition of $STree(T^{i-1})$ that leads to a leaf ℓ is called an *open transition* and is of the form $g'(s,(k,i-1)) = \ell$. When an update is made to a leaf $\ell = \overline{t_j \dots t_{i-1}}$ in $STrie(T^{i-1})$, a new state $\overline{t_j \dots t_i}$ is added. This implies that ℓ becomes an implicit state and it is replaced by $\ell' = \overline{t_j \dots t_i}$ in $STree(T^i)$. This will result in an update of g' with $g'(s,(k,i)) = \ell'$. For ℓ a leaf, and $g'(s,(k,p)) = \ell$ in $STree(T^i)$, p will always be equal to i. We can therefore represent all transitions to leaves as $g'(s,(k,\infty))$ where ∞ will be replaced by $|T|$ at the end of the algorithm. In this way it is not necessary to replace ℓ by ℓ' each time the tree is updated.

If we establish the *active point* $s_j = \overline{t_j \dots t_{i-1}}$ where $t_j \dots t_{i-1}$ is the longest suffix of $STree(T^{i-1})$ that occurs at least twice in T^{i-1}, and the *end point* $s_{j'} = \overline{t_{j'} \dots t_{i-1}}$ where $t_{j'} \dots t_{i-1}$ is the longest substring of T^{i-1} such that $t_{j'} \dots t_i$ is a substring of T^{i-1}. Both s_j and $s_{j'}$ are implicit or explicit states of $STrie(T^{i-1})$. We only have to see which updates of states and transition functions in $STrie(T^{i-1})$ on the boundary path from the active to the end point cause an update of their corresponding canonical reference pairs and

transition and suffix functions in $STree(T^{i-1})$. After this construction of $STree(T^i)$, state $g(s_{j'}, t_i)$ becomes the active point for $STree(T^i)$.

To make sure that each suffix of T is represented properly in $STree(T)$, we need to add a special *end marker* to string T. This character needs to be unique in the string, so that there will be no overlap between different suffixes in the tree. The character $ is an end marker that is commonly used for this purpose.

In Chapter 5 we will introduce several modifications on the data structure and construction of the suffix tree as suggested by Ukkonen. These modifications, i.e. *pruning* and *truncation*, are introduced to fit the problem of finding primers. The time complexity of the construction of the simple suffix tree is $\mathcal{O}(|T|)$. Since the suffix tree as described in the current section is an unmodified suffix tree, its complexity should be that of the modified trees, disregarding complexity introduced by these adaptions. A proof for the complexity of the suffix tree with modifications will be given in section 5.3. As an example of the simple suffix tree, the suffix tree for string `kokoskoek`$ is shown in figure 4.1. In this figure, the slightly curved edge is a suffix link.

Chapter 5

Modifications on the suffix tree

In the following sections we introduce several modifications on Ukkonen's data structure and algorithm to present suffix trees. The reason for the development of these extensions is to optimise the use of suffix trees for the problem of finding unique strings with certain features within a DNA-strand.

As mentioned in chapter 3 the DNA can be described as a set of strings. To make it possible for the suffix tree to contain several strings we can use a suffix tree over multiple strings as will be described in section 5.1.

Since we are only interested in primers up to a certain length it makes sense to only build trees up to a certain depth. These trees are introduced as truncated suffix trees in section 5.2.

Finally, we are only interested in strands that have certain features. In section 5.4 we introduce a modification of the algorithm, where we prune away nodes of the tree that have unwanted features that can be calculated through a non-decreasing function. If this function was not non-decreasing, some nodes that possess the correct features might be cut of and the shape of the tree will become unpredictable at some points which could affect the linear complexity of the Ukkonen algorithm. We will be able to use this type of tree to prune the truncated suffix tree on a subset of strands with unwanted GC-percentage and melting temperature.

5.1 A suffix tree over multiple strings

With a slight alteration of the Ukkonen algorithm, it is possible to build a suffix tree for multiple strings $\mathcal{T} = \{T_1, \ldots, T_n\}$. This modification was suggested earlier by Dan Gusfield in [Gus97]. The algorithm first creates the suffix tree $STree(T_1)$, and begins adding any new string T_i by resetting the active point to *root*. The resulting tree is called $STree(\mathcal{T})$ and is a representation of $\sigma(\mathcal{T})$ in space linear in length N. Here the reference pointers become (j, k, p), where j represents the j-th chunk, and k and p are the positions in that chunk.

A proof for both space and time complexity for this variation of the suffix tree will be given in section 5.3.

5.2 A truncated suffix tree

The idea of *truncated suffix trees* is introduced in [NAIP03]. In this section we will describe a formalisation of the idea to fit Ukkonen's suffix tree and we will prove that it remains linear proportional to the length of the input string in section 5.3.

Truncated suffix tree $STrunc(T, \Omega) = \{Q'', root, g'', f', c\}$ is a data structure that represents all substrings α of T where $|\alpha| \leq \Omega$. State $\perp$ is omitted in this variation of the suffix tree. The main reason for this is that there will be no need for it in the final implementation.

The leaves L of $STrunc(T, \Omega)$ are the states $\overline{\alpha}$ in $STrie(T)$ for which either $|\alpha| = \Omega$ or $\overline{\alpha} \in F$ and $|\alpha| \leq \Omega$. The other explicit states represented in $STrunc(T, \Omega)$ are the ancestors of L. The suffix function is only defined for a subset of the states contained in Q''.

A slight alteration is made to transition function g'. In the algorithm for the construction of $STree(T)$ as described in section 4.3, a leaf ℓ_i is created with transition function $g'(s(k, \infty)) = l_i$, in which ∞ will later be replaced by $|T|$, since state l_i represents suffix $t_i \ldots t_{|T|}$ of T. In $STrunc(T, \Omega)$ however, a leaf ℓ_i represents a substring $t_i \ldots t_{i'}$ with

$$i' = \begin{cases} i + \Omega - 1, & \text{for } i + \Omega - 1 < |T| \\ |T|, & \text{for } i + \Omega - 1 \geq |T| \end{cases}$$

and it is therefore cumbersome to replace every instance of ∞ with the correct value. Under the assumption that the length of string T is not

known in advance, it is possible to save space and not use more time than in the original algorithm. This can be achieved by redefining the transition function for some state s to some state r as $g''(s,(k,d)) = r$ in which d represents the length of the path from *root* to state r, instead of the index of the final position represented by r. With $g''(s'(k',d')) = s$ the string spelled out by the edge from state s to r is equal to $t_k \dots t_{k+d-d'-1}$. In this manner, for every leaf, d is set to Ω by default and can later be set to $|T|$ for those leaves representing the last Ω suffixes of T.

A *uniqueness label* c is added for states $\overline{\alpha}$ as

$$c(\overline{\alpha}) = \begin{cases} 0, & \overline{\alpha} \in L \text{ , } \alpha \text{ occurs only once in } T \\ 1, & \overline{\alpha} \in L \text{ , } \alpha \text{ occurs more than once in } T \\ \lambda, & \overline{\alpha} \notin L \end{cases}$$

This label will be used to determinate the uniqueness of every substring, and its exact use in obtaining information after the construction of the tree will be explained further in Chapter 6.

5.2.1 Algorithm

The algorithm for the creation of $STrunc(T,\Omega)$ is based on the algorithm that creates $STree(T)$. The main differences are the introduction of the uniqueness label c and the value of the active point s_j.

Initially $STrunc(T^0,\Omega)$ is created. From every $STrunc(T^{i-1},\Omega)$ with $1 \leq i \leq |T|$, $STrunc(T^i,\Omega)$ can be created, until in the last step $STrunc(T,\Omega)$ is constructed. The tree $STrunc(T^i,\Omega)$ is created from $STrunc(T^{i-1},\Omega)$ by adding all suffixes of string $t_{i-\Omega} \dots t_i$ (or $\sigma(t_{i-\Omega} \dots t_i)$) to it.

The active point s_j is found in a similar fashion as in Ukkonen's original algorithm. The tree $STrunc(T^{i-1},\Omega)$ is traversed starting at the active point and subsequently new states are created until end-point $s_{j'}$ is found. The active point of $STrunc(T^i,\Omega)$ will be $s_{j'}$ if $i - \Omega < j'$. If this is not the case j' will be increased by 1, namely $i - \Omega = j'$ has to be true, since the condition $i - 1 - \Omega < j$ and $j \leq j'$ also had to hold in $STrunc(T^{i-1},\Omega)$. The correct state $s_{j'+1}$ can then be found by proceeding as normal in the *update*-procedure for state $s_{j'}$, but avoiding all changes made to the tree.

Whenever a leaf ℓ is created as t_k-transition of $\overline{\alpha}$, the value of $c(\ell)$ is automatically set to 0, indicating that up to this point there is only one occurrence of a substring with prefix αt_k found. If the transition function for $\overline{\alpha}$ is set as $g''(\overline{\alpha'},(k',d')) = \overline{\alpha}$, it also states, that no other substring $\alpha t_{k+\Omega-d'-1}$ will

be found before the $k + \Omega - d'$-th character of T (if such a character exists). Now $c(\ell)$ will be set to 1 if the exact same substring is found again starting after the $k - d'$-th character.

If $t_{i+1-\Omega} \dots t_i = t_{i'+1-\Omega} \dots t_{i'}$, with $i < i'$, $s_{i'}$ will be the active point for the construction of $STrunc(T^{i'}, \Omega)$ from $STrunc(T^{i'-1}, \Omega)$. Namely if $t_{i+1-\Omega} \dots t_i = t_{i'+1-\Omega} \dots t_{i'}$, then $t_{i'+1-\Omega} \dots t_{i'}$ will be the longest substring of $t_{i'+1-\Omega} \dots t_{i'}$ that is also a substring of $T^{i'-1}$. When this is the case, label c will be update to $c(\overline{t_{i'+1-\Omega} \dots t_{i'}}) = 1$, indicating that the string it represents is not unique in T.

5.3 Truncated Ukkonen over multiple strings

The concepts described in sections 5.1 and 5.2 can be combined to form an algorithm which creates a truncated suffix tree over multiple strings. In the following sections an algorithm to create $STrunc(\mathcal{T}, \Omega)$ will be described followed by a proof of its complexity.

5.3.1 Algorithm

The algorithm for the creation of $STrunc(\mathcal{T}, \Omega)$ uses procedures `update`$(s, (j, k, i))$, `test-and-split`$(s, (j, k, i), t)$ and `canonise`$(s, (j, k, i))$, as represented below. These procedures are used to respectively update the tree with new character, to check whether a new node needs to be created and to find the next canonical pair. They are based on procedures with the same names described in [Ukk95]. With these modifications, the original procedures are extended to be used for multiple strings, decide whether a string is unique and to let leaf-nodes only represent strings of length Ω or shorter. For these last two modifications only procedure `update`$(s, (j, k, i))$ needed to be adjusted.

To emphasise the use of multiple strings, the parameter j (representing the string number) was added to all procedures, and the second parameter of g' was changed to (j, k, d), only for clarification.

For the details on the original procedures we refer to [Ukk95].

Procedure `update`(s, (j, k, i)) *with* $(s,(j,k,i-1))$ *the canonical reference pair for the active point*

---

1 **if** $i - k > \Omega$ ***or*** ($t_{j,i} = \$$ ***and*** $c(s) = 0$) **then**
2 $\quad$ $c(s) = 1$; $(s,k) \leftarrow$ canonise$(f'(s),(j,k,i-1))$;
3 oldr $\leftarrow$ root;
4 (end-point, r) $\leftarrow$ test-and-split$(s,(j,k,i-1),t_{j,i})$;
5 **while** ***not*** *(end-point)* **do**
6 $\quad$ create new transition $g'(r,(j,i,\Omega)) = r'$ where r' is a new state;
7 $\quad$ **if** *oldr* $\neq$ *root* **then** create new suffix link $f'(\text{oldr}) = r$;
8 $\quad$ oldr $\leftarrow r$;
9 $\quad$ $(s,k) \leftarrow$ canonise$(f'(s),(j,k,i-1))$;
10 $\quad$ (end-point, r) $\leftarrow$ test-and-split$(s,(j,k,i-1),t_{j,i})$;
11 **if** *oldr* $\neq$ *root* **then** create new suffix link $f'(\text{oldr}) = s$;
12 **return** (s,k)

Procedure `test-and-split`(s, (j, k, i), t)

---

1 **if** $k \leq i$ **then**
2 $\quad$ let $g'(s,(j,k',d)) = s'$ be the $t_{j,k}$-transition from s;
3 $\quad$ **if** $t = t_{j,k'+i-k+1}$ **then return** *(**true**, s)*
4 $\quad$ **else**
5 $\quad\quad$ replace the $t_{j,k}$-transition above by transitions
$\quad\quad$ $g'(s,(j,k',i-k)) = r$ and $g'(r,(j,k'+i-k+1,d)) = s'$
6 $\quad\quad$ where r is a new state;
7 $\quad\quad$ **return** *(**false**, r)*
8 **else**
9 $\quad$ **if** *there is no t-transition from s* **then return** *(**false**, s)*
10 $\quad$ **else return** *(**true**, s)*

Procedure `canonise`(s, (j, k, i))

---

1 **if** $i < k$ **then return** (s,k)
2 **else**
3 $\quad$ find the $t_{j,k}$-transition $g'(s,(j,k',d)) = s'$ from s;
4 $\quad$ **while** $d \leq i - k$ **do**
5 $\quad\quad$ $k \leftarrow k + d + 1$;
6 $\quad\quad$ $s \leftarrow s'$;
7 $\quad\quad$ **if** $k \leq i$ **then** find the $t_{j,k}$-transition $g'(s,(j,k',d)) = s'$ from s';
8 $\quad$ **return** (s,k)

Algorithm 5.1: Construction of $STrunc(\mathcal{T}, \Omega)$ for $\mathcal{T} = \{T_1 \dots T_n\}$ and $T_j = t_{j,1} \dots t_{j,n_j}\$$ for $1 \leq j \leq n$ a string over alphabet Σ. \$ is the end marker not appearing elsewhere in T_j.

1 create state *root*;
2 **for** $j \leftarrow 1, \dots, n$ **do**
3 $\quad s \leftarrow root; k \leftarrow 1; i \leftarrow 0;$
4 $\quad$ **while** $t_{j,i} \neq \$$ **do**
5 $\quad\quad i \leftarrow i + 1;$
6 $\quad\quad (s, k) \leftarrow \text{update}(s, j, (k, i));$
7 $\quad\quad (s, k) \leftarrow \text{canonise}(s, j, (k, i));$

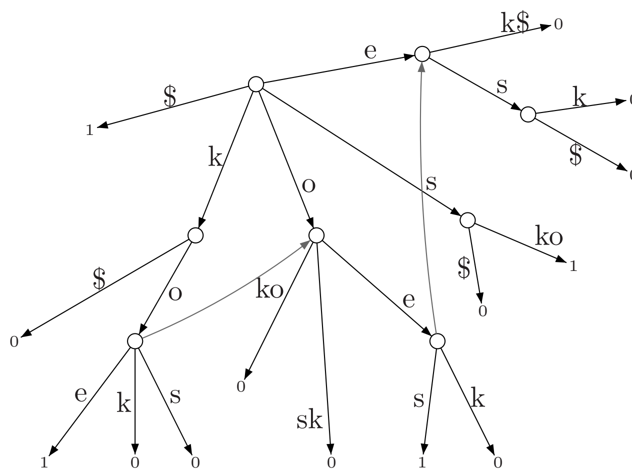

Figure 5.1: $STrunc(\{kokoskoek, koeskoes\}, 3)$

The final algorithm as used for this project is shown in 5.1. Figure 5.1 shows a graphical representation of $STrunc(\{kokoskoek, koeskoes\}, 3)$ (note that for convenience the indices (k, d) belonging to any edge are replaced by the string $t_k \dots t_{k-1+d}$).

5.3.2 Complexity

Let $\mathcal{T} = \{T_1, \dots, T_n\}$ a set of strings over alphabet Σ, and $N = \sum_{i=1}^{n} |T_i|$.

Theorem 5.1 *Algorithm 5.1 creates* $STrunc(\mathcal{T}, \Omega)$ *online with time complexity* $\mathcal{O}(N + n)$.

Proof. Algorithm 5.1 creates $STrunc(\mathcal{T},\Omega)$ by initially creating $STrunc(T_1^0,\Omega)$, and subsequently $STrunc(T_1^1,\Omega)$ through $STrunc(T_1^{n_1},\Omega)$, after which the end marker of string T_1 is added. Since, $STrunc(T_{i-1}^{n_1},\Omega) = STrunc(T_i^0,\Omega)$ (for $i > 1$), any $STrunc(T_j^0,\Omega)$ is created from $STrunc(T_{j-1}^{n_{j-1}},\Omega)$. The procedure is repeated for $j = 2,\ldots,n$ by repeatedly creating $STrunc(T_j^i,\Omega)$ from $STrunc(T_j^{i-1},\Omega)$ for $i = 1,\ldots,n_j+1$ (with $T_{n_j+1} = \$$) and resetting i to 1 for every new string that is digested. In the last step $STrunc(T_n^{n_n},\Omega) = STrunc(\mathcal{T},\Omega)$ is constructed. The algorithm is online, since only the characters read up to $T_{j,i}$ are necessary to construct $STrunc(T_j^i,\Omega)$. Every j-th traversal of the for-loop in algorithm 5.1 corresponds to adding $\sigma(T_j)$ to the tree, while every i-th repetition of the while-loop corresponds to the creation of $STrunc(T_j^i,\Omega)$ from $STrunc(T_j^{i-1},\Omega)$. This makes for a total of $N+n$ traversals of the while-loop.

To prove that the time complexity is linear to the sum of the total length of all strings and the number of strings, one needs to show that the number of steps necessary to create $STrunc(\mathcal{T},\Omega)$ is linearly proportional to $N+n$.

To show that this is the case we have to look at the procedures that contain loops. These are procedures canonise and update. Both of these procedures are called $N+n$ times in algorithm 5.1. First it will be made evident that the total number of times the while-loop in procedure update is traversed is linear proportional to $N+n$. At that point it can be concluded that the procedure canonise will be called a number of times which is linear proportional to $N+n$ (it is only called from the main algorithm and from the update-loop). Then it will be shown that the while-loop in procedure canonise is traversed in $\mathcal{O}(N+n)$, and it will become visible that $STrunc(\mathcal{T},\Omega)$ is created with algorithm 5.1 in $\mathcal{O}(N+n)$.

Every time the update-procedure is called for the creation of $STrunc(T_j^i,\Omega)$, it will traverse the suffix tree from the active point s_k to the end point $s_{k'}$. These states correspond to the substrings $t_k \ldots t_i$ and $t_{k'} \ldots t_i$ of T_j. Subsequently for the creation of $STrunc(T_j^{i+1},\Omega)$, the active point becomes $s_{k'}$. For any active point s_k and end point $s_{k'}$, $k \leq i$ and $k \leq k'$. Any traversal of the while-loop corresponds to increasing k by 1. For any string T_j, k can at most be increased n_j+1 times (including the end-marker) and there are therefore at most n_j traversals of the while-loop. Whenever $i = 1$ and $j > 1$, k is reset to 1 as well. This makes for a total of at most $N+n$ traversals.

The canonise function is called either directly from algorithm 5.1 or from the update procedure. In both cases, it will attempt to find the canonical reference pair for some state on the path from the active point to the end point of the tree that is being built. The number of times its while-loop is

traversed corresponds to the number of times an edge of the tree is traversed down. The depth of a state is defined as the length of the string spelled out by the path from the root to that state. Let r_i be the active point of $STrunc(T_j^i, \Omega)$ for $0 \leq i \leq n_j + 1$. The visited edges between r_{i-1} and r_i consist of a number of suffix links and one t_i-transition. Taking a suffix link decreases the depth by one and taking the t_i-transition increases it by one. The number of suffix links traversed is therefore $depth(r_{i-1}) - (depth(r_i) - 1)$ and the total number of visited states (not counting state r_i) is therefore $depth(r_{i-1}) - depth(r_i) + 2$. The total number of visited states is therefore $\sum_{j=1}^{n} \sum_{i=0}^{n_j+1} (depth(r_{i-1}) - depth(r_i) + 2) = \sum_{j=1}^{n} (depth(r_0) - depth(r_{n_j+1}) + 2(n_j + 1)) \leq 2(N + n)$. This makes for a total of $\mathcal{O}(N + n)$ link traversals.

The total time complexity is $\mathcal{O}(N + n)$. □

Theorem 5.2 *The size of $STrunc(\mathcal{T}, \Omega)$ has a worst-case complexity of $\mathcal{O}(min(N, |\Sigma|^{\Omega}) + \Omega n)$.*

Proof. The number of states in $STrunc(\mathcal{T}, \Omega)$ will never exceed the number of states in a tree which represents all possible strings of length Ω. In such a tree every branching state will branch into $|\Sigma|$ new states and the total number of explicit states will therefore be

$$\sum_{i=0}^{\Omega} \Sigma^i = \frac{|\Sigma|^{\Omega+1} - 1}{|\Sigma| - 1}$$

and the number of transitions will be

$$\frac{|\Sigma|^{\Omega+1} - 1}{|\Sigma| - 1} - 1$$

As stated in subsection 4.2.1, $STree(T)$ contains at most $2|T| - 2$ transitions. In total, n strings are added to the tree, with each string T_j having a total of n_j suffixes (excluding the end marker). The total number of suffixes is therefore N and $STrunc(\mathcal{T}, \Omega)$ contains at most N leaves, at most $N - 1$ branching states and $N - 2$ transitions.

Adding the fact that every string contains one end marker, a maximum of n leaves and n transitions is added to the tree. This results in a worst case of $min(2N - 2, \frac{|\Sigma|^{\Omega+1} - 1}{|\Sigma| - 1}) + 2\Omega n$ and therefore a space complexity of $\mathcal{O}(min(N, |\Sigma|^{\Omega}) + \Omega n)$. □

5.4 Pruning a tree on a non-decreasing function

The previous section describes how we can discard any substring from the tree that is longer than the strings we are interested in, while maintaining the functionality of the original suffix tree. It is also possible to discard strings from the tree that have other unwanted characteristics. As will be shown in this section, strings with characteristics that can be expressed as a non-decreasing function, can be *pruned* from the tree, without increasing space or time complexity.

Function Υ can be defined as follows.

Definition Let Υ be a function over all strings over an alphabet Σ with the property that for all strings $\alpha\beta \in \Sigma^*$:

i. $\Upsilon(\alpha) \leq \Upsilon(\alpha\beta)$ and $\Upsilon(\alpha) \leq \Upsilon(\beta\alpha)$.

ii. It is possible to calculate the following values in $|\beta|$ steps:
$\Upsilon(\alpha)$ from the value of $\Upsilon(\beta\alpha)$,
$\Upsilon(\alpha)$ from the value of $\Upsilon(\alpha\beta)$,
$\Upsilon(\beta\alpha)$ from the value of $\Upsilon(\alpha)$,
$\Upsilon(\alpha\beta)$ from the value of$\Upsilon(\alpha)$.

When there is only an interest in the strings α with $\Upsilon(\alpha) \leq X$, it is possible to adjust algorithm 5.1, so that a subset of the strings with a value for Υ higher than X will not be represented in the suffix tree.

5.4.1 Algorithm

After the creation of a new internal state $\overline{\alpha}$, $\Upsilon(\alpha)$ is calculated. When $\Upsilon(\alpha) > X$, the children of the state are removed from the tree. The state now becomes a leaf with the special feature that its depth is $|\alpha|$ instead of Ω. Also we do not have to update the uniqueness label $c(\overline{\alpha})$, since we are not interested in the uniqueness of a string that does not qualify $\Upsilon(\alpha) \leq X$. After this the algorithm proceeds as before.

As an example of the construction of a tree with this updated algorithm, figure 5.2 shows $STrunc^{\Upsilon}(\{kokoskoek, koeskoes\}, 3, 4)$, with function Υ is defined as $\Upsilon(\epsilon) = 0$, $\Upsilon(k) = 1$, $\Upsilon(e) = 2$, $\Upsilon(s) = 3$, $\Upsilon(o) = 4$ and $\Upsilon(\alpha\beta) = \Upsilon(\alpha) + \Upsilon(\beta)$. In this figure the nodes from which the children were cut off are depicted by filled circles.

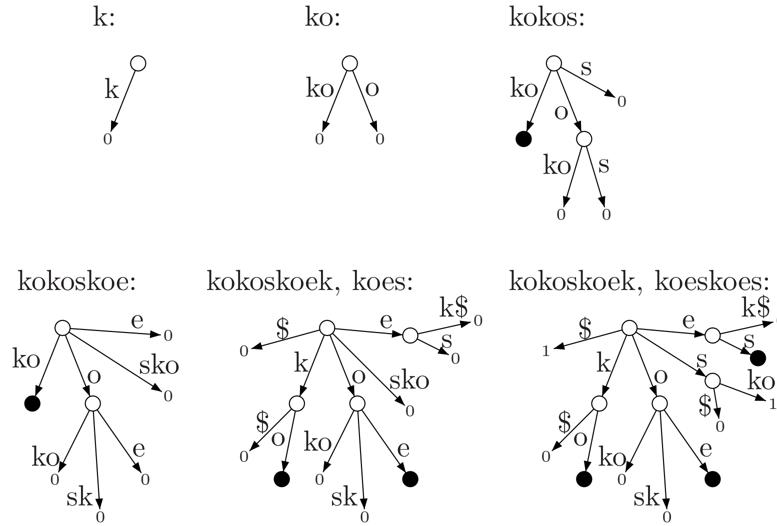

Figure 5.2: $STrunc^{\Upsilon}(\{kokoskoek, koeskoes\}, 3, 4)$ with Υ as on page 23

The new suffix tree resulting from the adjusted algorithm is called $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$. This new algorithm is correct in the sense that it will contain at least all substrings α (of maximum length Ω) with $\Upsilon(\alpha) < X$. Furthermore, this suffix tree variation is built in linear time and uses less space than $STrunc(\mathcal{T}, \Omega)$. For the next theorems we assume that Q'' are the explicit states of $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ and that its transition and suffix functions g'' and f' are defined only for some subset of the states of $STrunc(\mathcal{T}, \Omega)$.

5.4.2 Correctness

Theorem 5.3 *If α a substring of $\mathcal{T}$ with $|\alpha| \leq \Omega$ and $\Upsilon(\alpha) \leq X$, then there exists some explicit state $\overline{\beta}$ with α a prefix of β.*

Proof. Assume to the contrary that no such state β exists. This can have two reasons. Either there existed such a $\overline{\beta}$, but it was removed at some later point in the algorithm, or it was never inserted in the tree.

If a state $r = \overline{\beta}$ did exist, but was removed at some later point, then this must mean that there exists some prefix β' of β for which $\Upsilon(\beta') > X$ and an explicit state $r' = \overline{\beta'}$ was created. But according to the definition of function Υ, β' can not be a prefix of α and therefore α needs to be a prefix of β', which is a contradiction.

If no such state ever existed in the tree, then it was never added. This means that no string with prefix α is ever added to the tree. This can only be the case if for every string $T = t_1 \dots t_n \in \mathcal{T}$ and $\alpha = t_i \dots t_j$ (with $1 \leq i \leq j \leq n$) there exists a k with $i \leq k < j$ for which adding $\sigma(T^k)$ to the current suffix tree will result in a shift of the active point for $STrunc^{\Upsilon}(T^k + 1, \Omega, X)$ beyond implicit state $\overline{t_i \dots t_k + 1}$. This would imply that either, $\Upsilon(t_i \dots t_k) > X$ which is not possible according to the definition of Υ, or $t_i \dots t_k$ was not yet represented in $STrunc^{\Upsilon}(T^k - 1, \Omega, X)$ and a leaf $\overline{t_i \dots t_{i-1+\Omega}}$ is created. $\overline{t_i \dots t_{i-1+\Omega}}$ is an explicit state and α is a prefix of $\overline{t_i \dots t_{i-1+\Omega}}$. This is again a contradiction. □

In theorem 5.3, it was proven that no interesting information was lost from the suffix tree, when several substrings are pruned. Furthermore, we need to show that the suffix function f' is only defined for existing states and that any internal state $\overline{\alpha}$ (with $|\alpha| > 1$) has an outgoing suffix link.

Since we only update f' when a new internal state is created (and is not changed to a leaf immediately), and nodes are only pruned when they were already leaves or when they are added as the direct parent of a leaf, f' will always be well defined.

Furthermore, if $\overline{\alpha}$ with $\alpha = \beta\gamma$ and $\beta \in \Sigma$ a branching node in $STrunc^{\Upsilon}(\mathcal{T}_j^i, \Omega, X)$, than $\Upsilon(\alpha) \leq X$ and, because of the definition of Υ, $STrunc^{\Upsilon}(\mathcal{T}_j^i, \Omega, X)$ must contain a branching node $\overline{\gamma}$, with $f'(\overline{alpha}) = \overline{gamma}$.

5.4.3 Complexity

Theorem 5.4 *$STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ is created with time complexity $\mathcal{O}(N)$.*

Proof. The number of steps needed for the calculation of Υ is $\mathcal{O}(N)$. The substrings of $\mathcal{T}$ for which states are created are of the form $t_i \dots t_j$. For any subsequent event in which the Υ of some string $t_{i'} \dots t_{j'}$ needs to be calculated, $i < i'$, $j \leq j'$ and $i' \leq j'$ are true. According to the definition of Υ, the calculation of $\Upsilon(t_{i'} \dots t_{j'})$ from $\Upsilon(t_i \dots t_j)$ takes a maximum of $(i' - i) + (j' - j)$ steps. With I the number of internal nodes and $\overline{t_{i_k} \dots t_{j_k}}$ the k-th created internal node, the total steps needed is $\sum_{k=1}^{I}(i_k - i_{k-1}) + (j_k - j_{k-1}) = i_I - i_0 + j_I - j_0 \leq 2N$. At most $2N$ calculations are needed to calculated Υ for all strings for which new nodes are created.

For every newly created node $\overline{\alpha}$ with $\Upsilon(\alpha) > X$ there is at most one state that will be removed from the tree. When $\overline{\alpha}$ has children, it has exactly two: one that has just been created and one that was already there. For the child $\overline{\alpha\beta}$, which was created previously, $\Upsilon(\alpha\beta) \geq \Upsilon(\alpha) > X$ and $\overline{\alpha\beta}$

has therefore no children. Two states will be removed, which is one for every newly created state. The steps added to the algorithm add up to a complexity of $\mathcal{O}(1)$ per state of $STrunc(\mathcal{T}, \Omega)$.

The deletion of states from the tree does not add more steps to the total number of steps needed to traverse the tree during its creation. Whenever a state $\overline{\beta\alpha}$ (with $\beta \in \Sigma$) is deleted from the tree, then the newly created state was the parent state $\overline{\beta\alpha'}$ of $\overline{\beta\alpha}$. After that, the tree is traversed, across path p, to search for string α' as would have been the case if no deletion had occurred. Whenever string $\beta\alpha'$ is encountered again (and state $\overline{\beta\alpha'}$ still exists), the search for any string with prefix $\beta\alpha'$ is omitted, and instead path p will be crossed.

The updated algorithm remains linear. □

It is easy to see that the updated algorithm creates $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ in memory space less or equal to that needed for the creation of $STrunc(\mathcal{T}, \Omega)$.

5.4.4 Remarks

It can be seen that the length of a string $\alpha = t_i \dots t_j$ also has all the properties of Υ. This is a special case of a non-decreasing function though. Although we can calculate it in the number of steps suggested in the definition of Υ, the length of α can always be calculated in one step. Furthermore, the structure of the tree resulting from pruning by length is intuitively more obvious than that of pruning by other functions. We will therefore keep treating it as a separate operation.

It is possible to prune the tree by several functions that satisfy the definition of Υ. We can simply illustrate the correctness of this by using functions Υ_1 and Υ_2 with maximum values X_1 and X_2. If both of these functions are non-decreasing, pruning on both functions is equal to pruning each newly created internal node $\overline{\alpha}$ when condition $\Upsilon_1(\alpha) \leq X_1$ or $\Upsilon_2(\alpha) \leq X_2$ is not met. When either condition is not met, it will also not be met by any string that has α as a prefix or suffix. The number of steps to calculate a value as described in property ii in the definition of Υ becomes at most $|2\beta|$, which is still $|\beta|$ steps per function. Therefore, at most $2N$ calculations are needed per function and the time complexity remains the same.

Chapter 6

Counting substrings in suffix trees

The main interest for the use of suffix tree for this project was to discover short unique strands with certain properties in a large strand of DNA. The extensions introduced in chapter 5 were developed to target this problem with growing effectiveness. The following sections present procedures to discover the unique strings in different extensions to the suffix tree. The algorithms are ordered in descending space complexity.

6.1 In $STree(T)$

To count the number of occurrences of certain substrings in $STree(T)$, we can proceed as follows. For some string α over Σ, $g'(s,(k,k')) = r$ with $t_k \dots t_l = \alpha$ and $(r,(k',l))$ the canonical reference pair for some (explicit or implicit) state r'. If there is no such state r', and $g'(s,(k,k'))$ is undefined, α is no substring of T. If there is such a state r', then in $STrie(T)$, the path from *root* to r spell out the string α, and all paths from *root* to any descendant r'' of r', spells out some string $\alpha\beta$, with β a string over Σ. Leaf $l = \overline{t_i \dots t_{|T|}}$ descends from r' if and only if $t_i \dots t_{i+|\alpha|-1} = \alpha$. Therefore the number of leaves l in $STree(T)$ that descend from state r', is equal to the number of occurrences of string α in string T.

6.2 In $STree(\mathcal{T})$

The procedure of counting substrings in $STree(\mathcal{T})$ is carried out similar to that of section 6.1. Since in the case of multiple strings it is possible that a certain string is entered several times, all leaves in the tree need a counter. The sum of all counters in leaves l that descend from state r' with r' as in $STree(T)$, is then equal to the number of occurrences of α in T.

6.3 In $STrunc(\mathcal{T}, \Omega)$

The procedure of counting substrings in $STrunc(T, \Omega)$ is again similar to that for $STree(\mathcal{T})$, with the exception that only the uniqueness of substrings with a maximum length of Ω is defined by the uniqueness label c. The leaves of $STrunc(\mathcal{T}, \Omega)$ are equivalent to the leaves in $STree(\mathcal{T})$, with the difference that the leaves in $STrunc(\mathcal{T}, \Omega)$ have a depth value d set to Ω instead of an end-of-string value p. Since the start-position of the substring corresponding to the leaf and the end of the current string are known, the actual end-of-string value can easily be calculated. The uniqueness of a leaf can be read from its uniqueness label c. An internal state never represents a unique string.

6.4 In $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$

Leaf $\overline{\alpha}$ in $STrunc^{\Upsilon}(\mathcal{T})$ with a depth value d lower than Ω indicates that the corresponding substring has an Υ-value higher than X. We are therefore not interested in any string with prefix α and to exclude these from the final set of unique strings, any such string can be considered non-unique. Apart from this, uniqueness of a string is derived as for $STrunc(\mathcal{T}, \Omega)$.

Chapter 7

Transforming back to a string

The algorithm to create $STrunc(\mathcal{T}, \Omega)$ discussed in section 5.3 produces a tree from which the uniqueness of any substring of the input-file, of length Ω or less can be determined. The substrings that were pruned from the suffix tree with the algorithm to create $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ discussed in section 5.4 will be treated as non-unique strings. After executing the algorithm, the input-file is altered by adding a tag for every string to indicate whether or not it is unique.

7.1 Notation

With the problem discussed in [Lar05], where the length of the primer was fixed, the input-file was altered by adding a tag $\tau_i \in \{0, 1, 2, 3\}$ for each position i. The tags stand for 1, 2, 3 or 4 occurrences respectively. A similar approach is used for the extended problem. Now however τ_i will correspond to the uniqueness of strings. Here $\tau_i \in \{0, \ldots, \Omega\}$ and is equivalent to the statement that the string $t_i \ldots t_{i+j}$ with $j \in \{0 \ldots \Omega - 1\}$ is unique if and only if $j \geq (\Omega - \tau_i)$.

Every string corresponding with a string consisting of identified bases in the original file, is proceeded with a line of the form $[i, j, i', j']$. Here i and j represent the begin and end positions of the current substring in the total string with unidentified sections removed, while i' and j' represent the corresponding begin and end position in the original file. As an example, figure 7.1 displays the transformation of a file containing string `accgaattaaNNNNaaacg`,

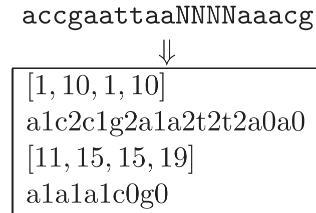

Figure 7.1: outputfile with $\Omega = 3$

to its output-file.

7.2 Complexity

Building the file is equivalent to a traversal through all the leaves of the tree in which the number of visits is equal to the total length of the strings represented by the tree.

In a non-truncated suffix tree, the maximum number of nodes visited is $4N$ with N the total length of all strings. For every string of length n_i represented by the tree, at most n_i nodes need to be traversed to reach the leaf presenting the entire string. From the father of the leaf the next leaf is reached by traversing one suffix link. The total number of transitions to reach the next leaf sum up to $2N$ for the entire traversal, as in the proof of theorem 5.1.

The only difference in the traversal of the truncated suffix tree is that the first leaf visited is reached in at most Ω transitions. The truncated tree is therefore traversed in at most $\Omega n + 3N$ visits with n the number of strings represented in the tree.

In the traversal of the suffix tree pruned with function Υ, the following holds: For every node $\overline{\alpha}$ that has been pruned to a leaf, all nodes representing strings $\overline{\beta\alpha}$ are leaves. By definition $\Upsilon(\beta\alpha) \geq \Upsilon(\alpha)$ and if $\overline{\beta\alpha}$ was an internal node, it must have been pruned after its creation. Therefore there are no internal nodes without an outgoing suffix link unless they are the children of the root. Any internal node $\overline{\alpha}$ can be reached from an internal node which is an ancestor of $\overline{\beta\alpha}$ in the same amount of steps as needed in the equivalent traversal in $STrunc(\mathcal{T}, \Omega)$. As in the proof of theorem 5.1, it is possible to show that the number of transitions from one leaf to another add up to $2N$. The number of visits is again at most $\Omega n + 3N$ and is probably lower – since the depth of traversal is lower for pruned areas – depending on the amount of nodes pruned with Υ.

Chapter 8

Counting DNA-strands with suffix trees

As stated in Chapter 3, it is possible to describe DNA as a set of strings $\mathcal{T} = \{T_{1,1}, \ldots, T_{1,a_1}, \ldots, T_{n,1}, \ldots, T_{n,a_n}\}$ over $\Sigma = \{A, C, G, T\}$. Here $T_{i,j} = t_{i,j,1} \ldots t_{i,j,L_{i,j}}$ represents substring j of chromosome i for $1 \leq i \leq n$ and $1 \leq j \leq a_i$ with n the number of chromosomes in $\mathcal{T}$, a_i the number of identified chunks of chromosome i and $L_{i,j}$ the length of chunk $T_{i,j}$.

We are looking for candidates for primers, which will have some maximum length. The data structures developed in Chapter 5 provide the structure needed to find these candidates with linear time and space complexity. The algorithm for $STrunc(\mathcal{T}, \Omega)$ is a suitable method to find all unique substrings of maximum length Ω in the chromosomes being examined, while $STrunc^{\Upsilon}(\mathcal{T}, \Omega, X)$ provides the possibility to further restrict the space needed to build a tree representing all interesting suffixes. In the following sections we will show how the suffix tree can be pruned on melting temperature and GC-percentage.

8.1 Pruning on melting temperature

It is not possible to filter the DNA-strand for melting temperature in advance when looking for primers of arbitrary length. It is however possible to restrict the growth of a suffix tree representing the DNA-string by considering the melting temperature corresponding to every internal node within the tree.

Below we will describe how the formula for the melting temperature as

developed by K.J. Breslauer et.al. in [BFBM86] of a strand of DNA can fit the restrictions of the non-decreasing function Υ as described in chapter 5.

The function `Tm` for the melting temperature in degrees Kelvin of some strand of DNA represented by α is defined as

$$\texttt{Tm}(\alpha) = \frac{\Delta H_d(\alpha) + \Delta H_i(\alpha)}{\Delta S_d(\alpha) + \Delta S_i(\alpha) + \Delta S_s(\alpha) + R \ln\left(\frac{C_T}{b}\right)} + 16.6[Na^+]$$

for strings α with $|\alpha| \geq 8$, where sums of enthalpy (ΔH_d) and entropy (ΔS_d) are calculated over all internal nearest-neighbor doublets, ΔS_s is the entropic penalty for selfcomplementary sequences, and ΔH_i and ΔS_i are the sums of initiation enthalpies and entropies, respectively. Here R is the gas constant (fixed at $1.987\frac{cal}{K \cdots mol}$) and C_T is the total strand concentration in molar units. Constant b adopts the value of 4 for non-self-complementary sequences or is equal to 1 for duplexes of self-complementary strands or for duplexes when one of the strands is in significant excess [BFBM86].

For any string α with $|\alpha| \geq 8$ it is possible to calculate a minimal value of any string $\alpha\beta$ with $|\alpha\beta| = \Omega$, by defining Tm^- as

$$Tm^-(\alpha) = \frac{\min_{\beta}^{\Sigma^{\Omega-|\alpha|}} \Delta H_d(\alpha\beta) + \min(\Delta H_i)}{\max_{\beta}^{\Sigma^{\Omega-|\alpha|}} \Delta S_d(\alpha\beta) + \max(\Delta S_i) + \max(\Delta S_s) + R \ln\left(\frac{C_T}{b}\right)} + 16.6[Na^+]$$

The values of $\Delta H_d(\alpha)$ and $\Delta S_d(\alpha)$ are the sums of the enthalpy and entropy of all neighbouring pairs. The value of the enthalpy and entropy of any possible pair is constant and can be looked up in a table. If $\alpha = a_1 \ldots a_i$, then $\Delta H_d(a_2 \ldots a_i) = \Delta H_d(\alpha) - \Delta H_d(a_1 a_2)$,$\Delta H_d(a_1 \ldots a_{i+1}) = \Delta H_d(\alpha) + \Delta H_d(a_i a_{i+1})$ and $\Delta S_d(a_2 \ldots a_i) = \Delta S_d(\alpha) - \Delta S_d(a_1 a_2)$, $\Delta S_d(a_1 \ldots a_{i+1}) = \Delta S_d(\alpha) + \Delta H_d(a_i a_{i+1})$. Both $\min_{\beta}^{\Sigma^{\Omega-|\alpha|}} \Delta H_d(\alpha\beta)$ and $\max_{\beta}^{\Sigma^{\Omega-|\alpha|}}$ can be calculated by adding $\Omega - |\alpha|$ times the minimal enthalpy and entropy to $\Delta H_d(\alpha)$ and $\Delta S_d(\alpha)$ respectively. The values $\min(\Delta H_i)$, $\max(\Delta S_i)$, $\max(\Delta S_s)$, $R \ln\left(\frac{C_T}{b}\right)$ and $[Na^+]$ are all constants.

Additionally for every $\alpha\beta \in \Sigma^{\Omega}$: $\texttt{Tm}(\alpha\beta) \geq Tm^-(\alpha)$ and therefore Tm^- has the properties of function Υ as described in section 5.4. If we are only interested in strings that have a melting temperature lower than X, this makes Tm^- suitable for the construction of $STrunc^{Tm^-}(\mathcal{T}, \Omega, X)$.

The function Tm^+ can then be defined as

$$Tm^+(\alpha) = \frac{\max_{\beta}^{\Sigma^{\Omega-|\alpha|}} \Delta H_d(\alpha\beta) + \max(\Delta H_i)}{\min_{\beta}^{\Sigma^{\Omega-|\alpha|}} \Delta S_d(\alpha\beta) + \min(\Delta S_i) + \min(\Delta S_s) + R\ln\left(\frac{C_T}{b}\right)} + 16.6[Na^+]$$

Since Tm^+ is a non-increasing function, $-Tm^+$ has the properties of function Υ as described in section 5.4. If we are only interested in strings that have a melting temperature higher than X, this makes Tm^+ suitable for the construction of $STrunc^{-Tm^+}(\mathcal{T}, \Omega, -X)$.

8.2 Pruning on GC-percentage

Since the maximum length of each substring is known in advance, it is possible to calculate the minimum percentage of G's and C's a string with prefix α will have with function `GC`.

$$\texttt{GC}(\alpha) = \frac{g(\alpha) + c(\alpha)}{\Omega}$$

Here $g(\alpha)$ and $c(\alpha)$ are the number of G's and C's in string α. For any α, $\texttt{GC}(\alpha)$ will be smaller or equal to the actual GC-percentage of any string that has α as a prefix and has a length of Ω or less. To be able do the total amount of calculations for the minimum GC-percentage of a string in $(O)(N)$ steps, we can simply keep track of the number of G and C nucleotides in the current string, by looking at the first character of the string before we decrease the sliding window and the last character after we increase it. This makes `GC` suitable for the construction of $STrunc^{\texttt{GC}}(\mathcal{T}, \Omega, X)$ where X is the maximum GC-percentage of interest.

In the same way the maximum GC-percentage can be calculated. The function

$$\texttt{AT}(\alpha) = \frac{|\alpha|}{\Omega} - \texttt{GC}(\alpha)$$

is the minimum percentage of A's and T's in strings of any string that has α as a prefix and has a length of Ω or less. $\texttt{AT}(\alpha)$ is suitable to use in the

construction of $STrunc^{\texttt{AT}}(\mathcal{T}, \Omega, 1-X)$, where X is the minimum percentage of G's in C's in a string to make it a candidate to be a primer.

8.3 Pruning on multiple functions

In summary of this chapter, it can be said that for any collection of chromosomes $\mathcal{T} = \{T_{1,1} \ldots T_{1,a_1} \ldots T_{n,1} \ldots T_{n,a_n}\}$, a pruned and truncated suffix tree $STrunc^{\{-Tm^+, Tm^-, \texttt{GC}, \texttt{AT}\}}(\mathcal{T}, \Omega, \{-Tm_{min}, Tm_{max}, GC_{max}, 1-GC_{min}\})$ can be created.

Chapter 9

Implementation

Although theoretically the Ukkonen algorithm is efficient in time and space complexity, in reality it requires both its input string and the resulting tree to be in main memory during its entire execution. The path that will be traversed through the tree is dependent on the input string, which is not known in advance and as a result no efficient ordering of the nodes is possible to assure spatial or temporal locality.

Every transition and suffix link needs to be represented by its memory location, and its indices. Although the space complexity of the individual transition and suffix links is treated as $\mathcal{O}(1)$ for convenience, in an actual implementation the complexity will be $\mathcal{O}(\log(V))$ with V the number of nodes in the tree. A link from one node to another will be represented by its memory-address which will have a length of $\lceil \log_2(V) \rceil$ bits. Apart from that, the label c needs to be defined for all leaves. For a set of n strings of total length N, there are at most $min(|\Sigma|^{\Omega+1}, N) + \Omega n$ leafs, at most $min\left(\frac{|\Sigma|^{\Omega}-1}{|\Sigma|-1}, N-1\right) + \Omega n$ internal nodes and at most $min\left(\frac{|\Sigma|^{\Omega+1}-1}{|\Sigma|-1} - 1, 2N-2\right) + 2\Omega n$ transitions.

If the transition function g' is defined for nodes s and r as $g''(s,(k,d)) = r$. The values of g'' need to be searchable for state s and character t_i and need to return state r and depth d. Each transition needs to be represented by at least $\left\lceil \log_2\left(min\left(\frac{|\Sigma|^{\Omega+1}-1}{|\Sigma|-1} - 1, 2N-2\right)\right)\right\rceil + \lceil \log_2(\Omega) \rceil$ bits of memory. Every internal node has at most one outgoing suffix link which needs $\left\lceil \log_2\left(min\left(\frac{|\Sigma|^{\Omega}-1}{|\Sigma|-1}, N-1\right)\right)\right\rceil$ bits. The counter function is defined for all leaves and needs to be represented by only 1 bit. The entire input string needs to remain in main memory and every character needs $\lceil \log_2(|\Sigma|) \rceil$ bits to be represented.

The entire length of human DNA is approximately 3×10^9 and we are interested in primers up to a length of approximately 30.

When we treat this as one string, the maximum amount of space needed to build a truncated suffix tree with $\Omega = 30$ for this tree is approximately $(6 \times 10^9) \times (\lceil \log_2(6 \times 10^9) \rceil + \lceil \log_2(30) \rceil) + (3 \times 10^9) \times (\lceil \log_2(3 \times 10^9) \rceil) + (3 \times 10^9) + (3 \times 10^9) \times (\lceil \log_2(4) \rceil) = 3.33 \times 10^{11}$ bits ≈ 38.7 Gigabytes.

Even when we reduce the maximum length to a number as low as 12, for which we only need approximately $\left(\frac{4^{13}-1}{3} - 1\right) \times \left(\left\lceil \log_2\left(\frac{4^{13}-1}{3} - 1\right)\right\rceil + \lceil \log_2(12) \rceil\right) + \left(\frac{4^{12}-1}{3} - 1\right) \times \left\lceil \log_2\left(\frac{4^{12}-1}{3} - 1\right)\right\rceil + (4^{12}) = 7.92 \times 10^8$ bits ≈ 95 Megabytes to represent the tree, we still need 6×10^9 bits ≈ 715 Megabytes to represent the character string.

Because the character string needs to remain in main memory, another problem arises. The Ukkonen algorithm is in theory online, so for every character that is added to the string, new memory needs to be allocated. For large strings this can result in a huge slow-down of the algorithm.

The following sections describe an implementation that we performed for this project.

9.1 Building the suffix tree

The program `suffixtreetrunc.c` is an implementation of the truncated and pruned suffix tree for multiple strings $STrunc^{\{-Tm^+, Tm^-, \mathtt{GC}, \mathtt{AT}\}}(\mathcal{T}, \Omega, \{-Tm_{min}, Tm_{max}, GC_{max}, 1 - GC_{min}\})$ as described in chapter 8.

The following table lists the input-variables for `suffixtreetrunc.c`

option	function	default value
-i	name of the input file	`chr.fa`
-o	name of the output file	`output`
-m	maximum length of interest	3
-g	minimum percentage of G·C-pairs	0
-h	maximum percentage of G·C-pairs	100
-t	minimum melting temperature in degrees Celsius	0
-u	maximum melting temperature in degrees Celsius	100
-d	concentration of annealing primers in mol/ml	50
-e	salt concentration in mol/ml	50

The algorithm used in `suffixtreetrunc.c` is based on the configuration described in the following subsections.

9.1.1 Nodes

For the sake of simplicity, memory-space is allocated for each node for the transition and suffix functions and the uniqueness label (f, g and c). To use constant space for each node, each node has one pointer to its first created child and one pointer to its brother. The correct child needed for traversing a transition link is therefore possibly found by going through a number of brothers of the first child. This results in a total of 19 bytes per node.

The nodes and functions related to updating them are defined in `suffixtree.c` and `suffixtree.h`.

9.1.2 Character string

The characters in the input file are read one by one and the character string is kept in main memory in a previously allocated buffer.

After the tree has been built, it will be traversed to search for every substring of length Ω and the last $\Omega - 1$ suffices of every substring. For every character and its respective counter, as described in chapter 7, one `char` is written to the output file, in which the first 2 bits represent the character at that position (a = `0x00`, c = `0x01`, g = `0x10`, t = `0x11`) and the last 6 bits represent the counter. This restricts the value of Ω in this implementation to a maximum of 63, and the size of the output file will be approximately equal to that of the input file.

The reading and writing of characters is defined in `dnafile.c` and `dnafile.h`.

9.1.3 Filtering

The functions Tm^{+}, Tm^{-}, `GC` and `AT` as described in chapter 8, are implemented with restrictions (ii) in the definition Υ in chapter 5. This requires the current values for the GC-count, enthalpy and entropy values for the current suffix $t_j \dots t_{i+1}$ to be updated with every increase of either j or i in time $\mathcal{O}(1)$. The functions are defined with this characteristic in `functions.c` and `functions.h`. The functions representing the melting temperature are loosely based on the equivalent functions developed in [Lar05].

These functions do not guarantee the filtering of all unwanted substrings from the tree. The output file still needs to be filtered for wrong Tm^{+}, Tm^{-}, `GC` and `AT` values.

9.2 Reading from an output file

To interpret the output file created by `suffixtruncplus.c` in the format as described in chapter 7 the program `search.c` was created.

9.2.1 Search

The following table lists the input variables for `search.c`

option	function	default value
-i	name of the input file	`chr.fa`
-l	length of interest	3
-g	minimum GC-percentage	20
-h	maximum GC-percentage	80
-t	minimum melting temperature in degrees Celsius	60
-u	maximum melting temperature in degrees Celsius	63

The values for the length of interest, the GC-percentages, and melting temperatures are restricted to those used to build the suffix tree represented in the file being examined. The uniqueness of all strings of some length ($\leq \Omega$) can be found as described in chapter 6. For every unique string the values for Tm^{+}, Tm^{-}, `GC` and `AT` are calculated with the aid of `functions.c` and `functions.h`. In this manner it can be decided which substrings could be appropriate to be used as primers.

Chapter 10

Experiments

The programs described in chapter 9 were used to obtain information about the speed-up and decrease in memory resulting from both the restriction of the depth and the pruning according to function Υ. This information will be presented and discussed in the following sections.

In all of the discussions, the emphasis needs to be on any improvement in the time and space needed for the construction of the tree. As discussed in chapter 9, the amount of memory needed is the bottleneck for Ukkonen's suffix tree algorithm. When a considerable amount of memory can be saved at the cost of a slow-down, this is preferred to a large speed-up which has an increase in necessary memory as its result.

10.1 Input file

The input files used for the experiments for this project were taken from the genome database of the University of California Santa Cruz [UCS]. This database contains tables in the FASTA format, describing the chromosomes of several species. For the experiments described in this chapter only the human genome was examined.

A table representing a chromosome is a file containing a string over alphabet $\Sigma = \{a, c, t, g, A, C, T, G, n, N\}$ in which every character represents the nucleotide on that position. Both n and N indicate the absence of knowledge about the nucleotide at that position. The capital letters A, C, T, G represent that the nucleotide on that position is part of a sequence of up to 12 nucleotides that is repeated several times. Since for the problem assessed in this project we look at unique sequences of variable lengths (often more

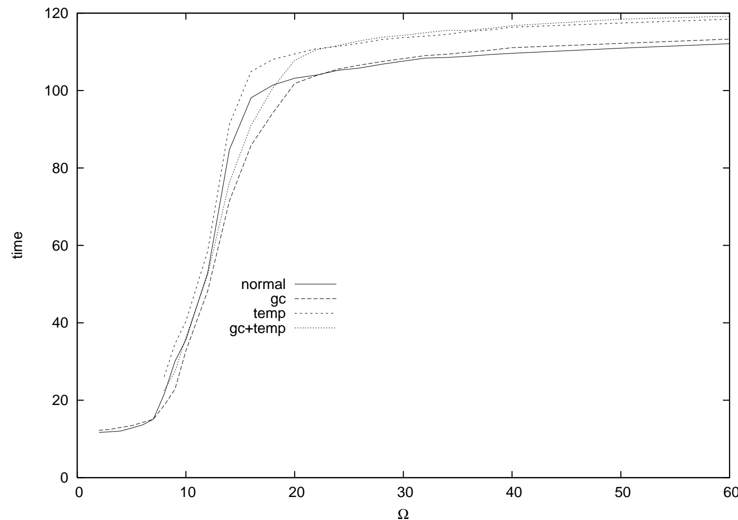

Figure 10.1: Seconds of CPU-time needed to build a tree with depth Ω for the 21-th human chromosome

than 12), it is not certain whether or not theses regions will be interesting. Because of this, the capital letters A, C, T and G will be treated as a, c, t and g respectively.

For the results shown in the following pages, we used a Pentium IV machine with 3.20 GHz and 3.2 Gigabytes of available memory. Since each node is represented using 19 bytes of memory, we could store a full suffix tree for a string of length at most 8.8×10^7 characters. Alternatively, we could for example store a truncated suffix tree of depth at most 13, which leaves room for a 1.7×10^9 character string. In the experiments done here, the 21-th human chromosome was used, which is represented in the genome database of the University of California Santa Cruz [UCS] in file `chr21.fa` and contains approximately 4.7×10^7 characters (3.4×10^7 when we exclude the regions with N-characters), and will therefore easily fit in main memory.

10.2 Time

The time needed to build both the tree and the output file was measured for different values for the maximum depth Ω, ranging from 2 to 60. The trees were pruned with respectively no function, functions `GC` and `AT` with, for both, maximum value .55, functions $-Tm^+$ and Tm^- with maximum values $-25°$C and $30°$C and finally for all 4 functions. In figure 10.1 the CPU-time in seconds to build the tree for each of this variations is shown.

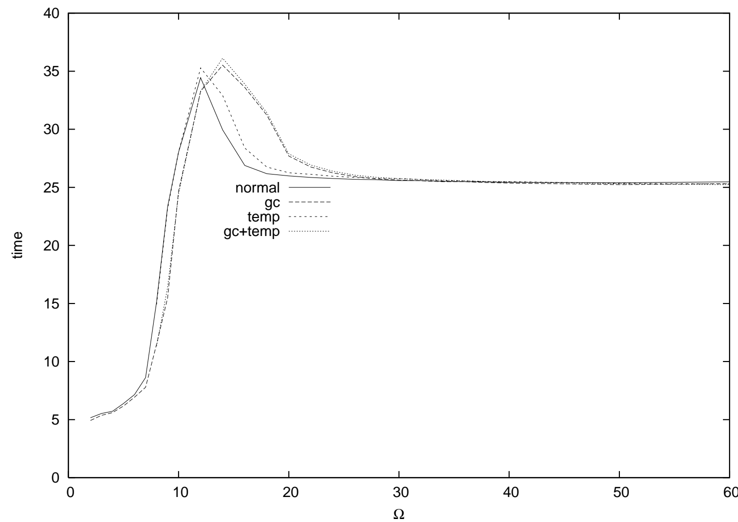

Figure 10.2: Seconds of CPU-time needed to build a file representing a tree with depth Ω for the 21-th human chromosome

The time needed to write to the output file is depicted in figure 10.2.

The relative speed-up of pruning according to Υ compared to non-pruning is shown in figure 10.3. The times compared here are the sums of the time to build both the tree and the output file.

The time needed to build the truncated tree stabilises at large values of Ω and the program executes noticeably faster for $\Omega \leq 14$.

In the graph showing the time needed to build the output file as shown in figure 10.2 an obvious decrease in time is noticeable for $\Omega \geq 12$. This can be explained by the amount of searches needed to find a child node. Before a transition to a child node can be used, the correct child node needs to be determined. In Chapter 7 we treated search operations for the correct child of a node as if they would take approximately 1 step. Since most strings of short length can be found in the large file, most internal nodes with a small depth, will likely have $|\Sigma| = 4$ children. This is not by definition the case for nodes with a larger depth. Since the traversal of the tree in order to find all leaves takes place in the lower regions of the tree, it is likely that more transition options need to be considered for smaller Ω, where nodes in the lower region of the tree mostly have 4 children. More specifically, when a node has 2 children, it can be expected that finding the correct child will take $\frac{1}{2} \times 1 + \frac{1}{2} \times 2 = 1\frac{1}{2}$ steps, while in a node with 4 children, the expected number of steps is $\frac{1}{4} \times 1 + \frac{1}{4} \times 2 + \frac{1}{4} \times 3 + \frac{1}{4} \times 4 = 2\frac{1}{2}$.

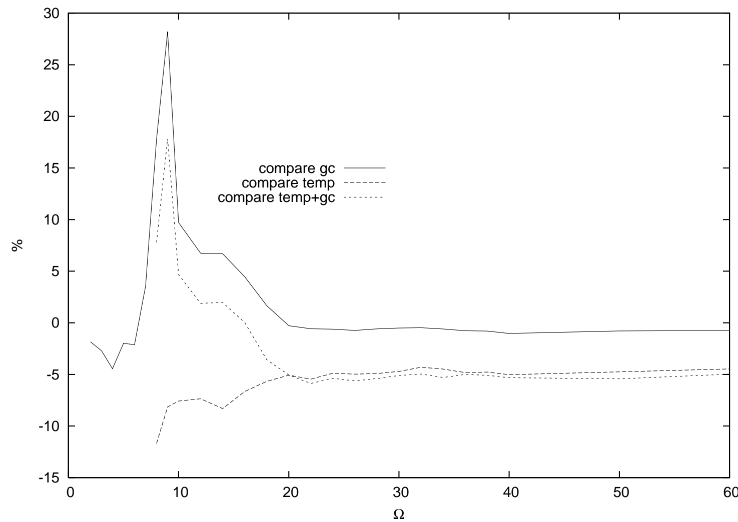

Figure 10.3: Overall speed-up for every Υ

Pruning on GC-percentage causes a large speed-up for $\Omega < 20$, which is probably caused by a decrease in number of nodes. The decrease in nodes by pruning on melting temperature does not outweigh the time needed for its calculation for those values of Ω. Overall the pruning functions create a small slow-down for larger values.

10.3 Number of nodes

With the same parameters as in the previous section the number of nodes in the resulting tree were counted. This gives an indication of the amount of memory needed for each suffix tree variation.

The graph in figure 10.4 shows the size of the tree for the different values of Ω. A comparison was again made between the different functions Υ and the decrease in nodes is drawn in figure 10.5.

It can be assumed that the space required to complete building the entire suffix tree is approximately that needed to build the truncated suffix tree with maximum depth 13 or more. There are $4^{13} \approx 6.7 \times 10^7$ possible strings of length 13 over alphabet Σ. Since the file examined here contains approximately 3.4×10^7 characters in identified regions, it can be expected that most of its substrings of length 13 or more are unique. Any leaf representing substring α in a truncated suffix tree with counter $c(\alpha) = 0$ will also be a leaf in the non-truncated suffix tree for the same string. DNA is not a

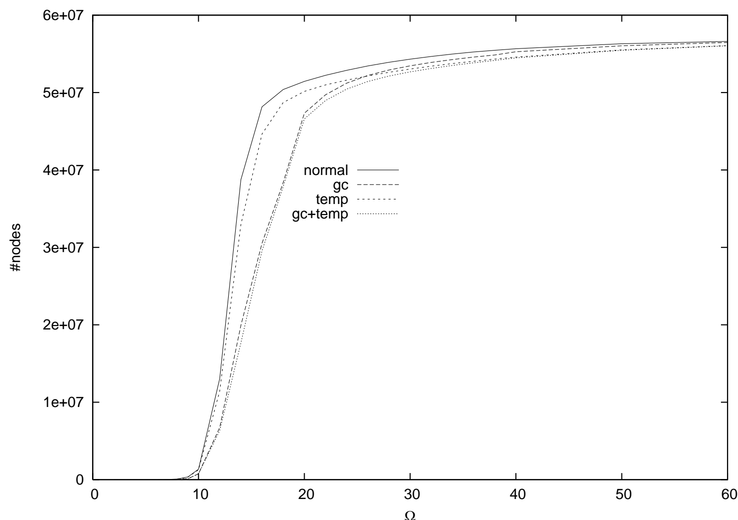

Figure 10.4: Number of nodes in a tree with depth Ω for the 21-th human chromosome

random strand and in general contains large regions of repeating patterns. As can be seen in figure 10.4 only at approximately $\Omega = 40$ the number of nodes starts to stabilise. Although this is an advantage on number of nodes, the truncation of the suffix tree has introduced the counter function c which needs to be defined for every leaf and will result in a slightly larger use of memory. A definite improvement in memory can be seen for $\Omega \leq 20$.

Pruning the tree with both the GC functions `GC` and `AT` as the temperature functions Tm^+ and Tm^- causes a large number of nodes to be cut of. This is especially the case for $\Omega < 20$ where a large decrease in number of nodes is noticeable, while the decrease is only small for $30 < \Omega < 60$.

It seems as if the GC functions account for a larger amount of pruned nodes than the temperature functions. This is chiefly the case for lower values of Ω while even for large values of Ω a small amount of nodes is still pruned with the melting temperature. Note however, that the values for minimum and maximum temperature (25° and 30°C) are unrealistic. These values were chosen to enhance the effect of the temperature function and give a clearer contrast against the other functions. As mentioned in chapter 2 values between 55° and 60°C are more useful when searching for primers. The effect of using these as extreme values will have a smaller effect on the decrease of time and space needed for the tree, than the values used here.

Furthermore, we need to note that there is a correlation between the melting temperature of a strand and its GC-percentage. It can be seen in [BFBM86]

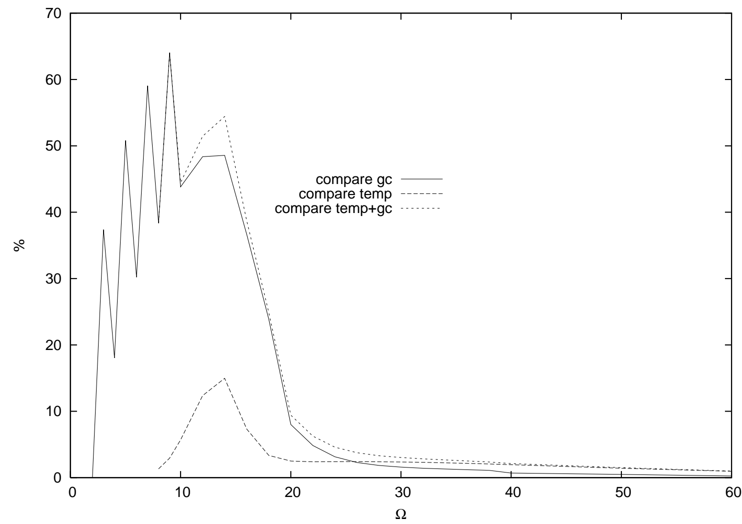

Figure 10.5: Percentage of decrease in number of nodes for every Υ

that G and C nucleotides cause the melting temperature to rise more than A and T nucleotides. This explains why we only see a small increase in percentage of number of nodes in figure 10.5 when we prune on both temperature and GC-percentage.

Chapter 11

Conclusions

When looking for short unique substrings of variable length within a string over some known alphabet, the truncated suffix tree is an efficient method to identify such strings. If the suffix tree representing them can fit in main memory, its creation and analysis can be conducted in linear time.

If, additionally, these substrings need to satisfy some non-decreasing function it is possible to reduce the size of the tree even more by pruning its nodes during creation. The total algorithm remains linear if the following holds. For any string for which the value of the function is known, if we add or take one character from the string, the value of the function for the new string can be calculated in $\mathcal{O}(1)$.

It is possible to apply these methods on DNA where there is an interest in substrings up to a maximum length of around 50 and the tree needs to be restricted on the functions for the GC-percentage and the melting temperature. Especially when a large amount of DNA material is analysed with this method, the truncation for a maximum tree depth of 20 can give a large decrease in memory space and CPU time. Pruning on both the GC-percentage and temperature functions can give a large further decrease in both time and memory. The decrease in time achieves its peek between tree depths of 10 and 20, while that of the space lies between 10 and 30. For the Ukkonen algorithm where memory is the bottleneck, the decrease in memory should count more then the decrease in time.

Since the ideal length for a primer lies between approximately 10 and 30 pruning and truncating the suffix tree provides a suitable method to discover unique substrings.

Chapter 12

Further research

As shown in chapter 9 the maximum possible length of the string represented by the tree is very much dependent on the amount of memory available. When the string exceeds this length, it will no longer be possible to hold the entire tree in main memory. Because of the lack of spatial or temporal locality, the algorithm can no longer be executed in linear time. For further research it could be beneficial to find a method to build the suffix tree in a manner which does not require it to reside completely in memory, possibly at the cost of a large time complexity. With such an algorithm the truncation and pruning of the tree could still result in significant improvements.

The methods discussed here are not restricted to the search for primers. In future research a further analysis of the pruning algorithm can be made without the emphasis on its possible application.

Bibliography

[AJL$^+$02] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell, fourth edition.* Garland Science, 2002.

[BFBM86] K. J. Breslauer, R. Frank, H. Blöcker, and L. A. Marky. Predicting DNA duplex stability from the base sequence. *Proc Natl Acad Sci U S A*, 83(11):3746–3750, June 1986.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University press, 1997.

[Lar05] Jeroen Laros. Unique factors in the human genome. Master's thesis, Leiden University, May 2005.

[McC76] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, 1976.

[NAIP03] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Compututer Science*, 304(1-3):87–101, 2003.

[UCS] University of California Santa Cruz UCSC. *Genome Informatics.* http://genome.ucsc.edu/.

[Ukk95] Esko Ukkonen. online construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.