Data structure for representation of maximal repeats in strings

Michal Linhard

Bratislava 2007

Data structure for representation of maximal repeats in strings

Diploma thesis

Michal Linhard

Thesis advisor: Mgr. Tibor Hegedüs

Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science

Branch of study: Informatics

Bratislava 2007

Abstract

In this diploma thesis we present data structure for representation of maximal repeats in strings - R3 tree, based on well known data structure - suffix tree. It requires $O(n)$ space and it can be constructed in $O(n)$ time and space for string of length n over constant-sized alphabet. We formalize repeat in string S as triple (p_1, p_2, l), where p_1, p_2 are two distinct positions in S and l is the length of the repeat. We formulate query for maximal repeats in S in the form of the function $findPairs(p_1, k, S)$ that returns all pairs (p_2, l) such that (p_1, p_2, l) is maximal repeat in S with $l \geq k$. R3 tree allows computation of findPairs queries in optimal time $O(z)$, where z is the number of found pairs. We also describe design and functionality of R3lib – library written in C, for finding maximal repeats in arbitrary binary data, that works with proposed structure.

Keywords: maximal repeats, representation of repeats, repetition in strings

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

Michal Linhard

Contents

1 Introduction **3**

2 Definitions and notation **5**

2.1 Basic notions 5

2.2 Suffix tree and lcp-interval tree 6

2.3 Reverse lcp-interval 11

3 R3 Tree **14**

3.1 LC-bucket tree 14

3.2 Non-optimal findPairs query on LC-bucket tree 16

3.3 Union trees 18

3.4 R3 tree 21

3.5 Optimal findPairs query on R3 tree 24

3.6 Properties of R3 tree 26

4 R3 tree implementation **28**

4.1 Representation of Union forest 28

4.2 Representation of R3 tree 29

4.3 Construction of R3 tree 30

4.3.1 Phase 1: suffix array and lcp-table construction 33

4.3.2 Phase 2: bottom-up traversal (LCP, BP, BPTABLES, UF, FD) 33

4.3.3 Phase 3: UP1 construction 36

4.3.4 Phase 4: top-down traversal (LC1, UP2) 37

4.3.5 Phase 5: MAP construction 37

A R3lib documentation **39**

B Performance results **43**

Bibliography **45**

List of figures **46**

Chapter 1

Introduction

The most known motivation for maximal repeat algorithms comes from bioinformatics. The computation of maximal repeats in strings plays an important role in the analysis of genomic sequences. In general, this area stimulates majority of research in area of string algorithms today. There are several other motivations for finding duplication in any data. Repeat discovery may help avoiding redundancy and can be useful in text analysis. For example, it is a good practice to avoid duplication in program source code because of the danger of bug fixes being applied to one copy but not all the others.

There are algorithms and software tools for finding all maximal repeats in a string. Optimal algorithm for finding all maximal repeats was first described in [1]. This algorithm is based on suffix trees and finds all maximal repeats in $O(n.log|\Sigma| + z)$, where n is length of the string, $|\Sigma|$ is size of the alphabet and z is number of maximal repeats (output size). A space efficient version of this algorithm using suffix arrays is described in [2]. There are tools that can efficiently find maximal repeats in genome sequences, for example Vmatch[1] (new version of REPuter) and also recent version of MUMmer[2].

Maximal number of all maximal repeats in a string S of size n is $O(n^2)$. Some applications may occur, where we don't want to see all maximal repeats at once, but interactively analyze data or text and see only maximal repeats starting at position in currently viewed segment. Our approach is to build a data structure representing all maximal repeats in the data, that could answer such queries quickly. It turns out, that such structure requires only linear space and also can be constructed in linear time and space.

We decided to call our data structure R3 tree. The part 'R3' comes from Repeat Report Representation, as this structure can effectively replace "repeat report" (list of all maximal repeats) in some applications.

The main task of this diploma thesis was to develop a library that supports this approach to maximal repeats. Our library, written in C, is called R3lib and the latest version can be found on the R3lib project home page at `http://michal.linhard.sk/r3lib`.

The contents of this thesis are organized as follows. In *Chapter 2* we define the basic terms and data structures. Suffix tree and lcp-interval tree will be defined in terms of graph theory and isomorphism of these two structures will be established. This is because R3 tree

[1]http://www.vmatch.de/

[2]http://mummer.sourceforge.net/

will be conceptually defined in terms of suffix tree but implementation in R3lib works with suffix array and lcp-interval tree. This approach - replacing suffix trees by enhanced suffix arrays - was introduced in [2] and this work will be referenced many times throughout our text.

Chapter 3 starts by introducing LC-bucket tree. This structure is defined, so that we can demonstrate problems that have to be solved for findPairs query to run in optimal time. Then we define R3 tree and show how it deals with these problems.

In *Chapter 4* we describe data structures that are used in R3lib library to represent R3 tree and we show how R3lib constructs R3 tree in this representation.

Chapter 2

Definitions and notation

2.1 Basic notions

Let Σ be finite ordered alphabet. We will use symbols $a, b, \ldots$ for elements of the alphabet. We suppose $|\Sigma|$ is constant. Σ^* is the set of all strings over Σ. Let $>_L$ denote lexicographic ordering on Σ. We will use symbols $S, S_1, S_2, x, y, \ldots$ for strings. $|S|$ denotes length of string S. Reverse of string S is denoted S^R. We write $S[i]$, where $0 \leq i < n$, to refer to i-th character of S. We define $S[-1] = ¢$ and $S[n] = \$$, where $¢, \$$ are special symbols not occuring in Σ. $\forall a \in \Sigma : \$ > a$. $S[i..j]$, where $0 \leq i \leq j \leq n$ refers to substring of S starting at position i and ending at position j. For $0 \leq i < n$, substring $S[i..n-1]$ is called suffix and substring $S[0..i]$ is called prefix. The fact that S_1 is prefix of S_2, is denoted $x \sqsubseteq y$ and the fact that x is suffix of y is denoted $y \sqsupseteq x$. Set of natural numbers smaller than n is denoted $N_n = \{0, 1, \ldots, n-1\}$, i-th suffix with end marker is denoted $suffix_S(i) = S[i..|S|]$, left context of i-th suffix is denoted $LC_S(i) = S[i-1]$.

For two strings x, y we define $lcplen(x, y)$ to be length of their common prefix.

A triple (p_1, p_2, l) is called **repeat** in string S if $0 \leq p_1 + l \leq |S|$, $0 \leq p_2 + l \leq |S|$, $p_1 \neq p_2$ and $S[p_1 \ldots p_1 + l - 1] = S[p_2 \ldots p_2 + l - 1]$. Repeat (p_1, p_2, l) is called **left maximal** if $LC_S(p_1) \neq LC_S(p_2)$ and **right maximal** if $S[p_1 + l] \neq S[p_2 + l]$. A repeat is called **maximal** if it's left and right maximal.

On figure 2.1, we can see example of maximal repeat $(1, 25, 7)$. Our queries for maximal repeats in string S will have form of $findPairs(p_1, k, S)$, where findPairs is a function that returns the set of all pairs (p_2, l) such that (p_1, p_2, l) is maximal repeat in S with $l \geq k$. For example the query $findPairs(0, 4, S)$ for string from figure 2.1 would return the set $\{(4, 5), (11, 4), (28, 4)\}$.

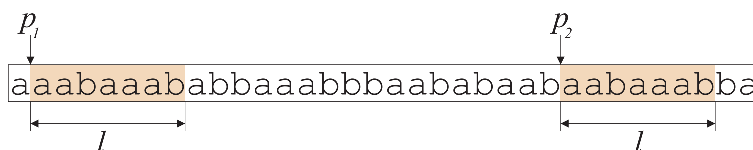

Figure 2.1: Example of maximal repeat

Tree T is a triple $(V, E, root)$ where V is set of nodes, $root \in V$ is the root node, $E \subseteq V \times V$ is set of edges. For all nodes $v \in V \setminus \{root\}$ there is exactly one node $parent_T(v) \in V$ such that $(parent_T(v), v) \in E$. For a node v we define $Children_T(v) = \{u|(v,u) \in E\}$, $Desc_T(v) = \{u|(v,u) \in E^+\}$, where E^+ is transitive closure of E. Depth of a node is defined as follows: $depth_T(root) = 0$, $depth_T(v) = depth_T(parent_T(v)) + 1$ for $v \in V \setminus \{root\}$. Lowest common ancestor of two nodes will be denoted $LCA_T(v_1, v_2)$.

We divide set of nodes V into leaves V_L, and internal nodes V_I. We divide set of edges E into internal edges $E_I = E \cap (V_I \times V_I)$ and leaf edges $E_L = E \cap (V_I \times V_L)$.

For a tree T, we use following symbols. $V(T)$ is set of nodes, $E(T)$ is set of edges, $root(T)$ is the root of the tree, $V_I(T)$ is the set of internal nodes, $V_L(T)$ is set of leaves, $E_I(T)$ is the set of internal edges, $E_L(T)$ is the set of leaf edges. $T_I(T) = (V_I(T), E_I(T), root)$ will denote internal part of suffix tree. E^+ will denote transitive closure of relation E, whenever E is used as symbol for set of edges.

2.2 Suffix tree and lcp-interval tree

Suffix tree for a string S is a 5-tuple $T = (V, E, root, label, prefix)$. First three components form a tree $\dot{T} = (V, E, root)$. We define the same symbols for suffix tree as we defined for tree: $V(T) = V$, $E(T) = E$, $root(T) = root$, $V_I(T) = V_I(\dot{T})$, $V_L(T) = V_L(\dot{T})$, $E_I(T) = E_I(\dot{T})$, $E_L(T) = E_L(\dot{T})$, $Chidren_T = Children_{\dot{T}}$, $Desc_T = Desc_{\dot{T}}$, $parent_T = parent_{\dot{T}}$, $depth_T = depth_{\dot{T}}$, $LCA_T(v_1, v_2) = LCA_{\dot{T}}(v_1, v_2)$.

$label_T : E \to \Sigma^+$ is an edge-labeling function, that labels each edge of the tree T by some non-empty string. If $label_T(e) = ax$ for some $x \in \Sigma^*$, we call e an a-edge.

$prefix_T : V \to \Sigma^*$ is a map from nodes to strings. For a node $v \in V \setminus \{root\}$ and the path $root = v_0, v_1, \ldots, v_k = v$ we define $prefix_T(v) = label_T(v_0, v_1) label_T(v_1, v_2) \ldots label_T(v_{k-1}, v_k)$, $prefix_T(root) = \varepsilon$.

Suffix tree T satisfies following conditions

1) $V_L(T) = N_{|S|+1}$
2) $\forall i \in V_L(T) : prefix(i) = suffix_S(i)$
3) $\forall v \in V_I(T) : |Children_T(v)| \geq 2$
4) $\forall v \in V_I(T) : \forall i \in Desc_T(v) \cap V_L(T) : prefix_T(v) \sqsubseteq prefix_T(i)$
5) $\forall v \in V_I(T) : \big((v,u) \in E \wedge (v,w) \in E \wedge (v,u) \text{ is } a\text{-edge } \wedge (v,w) \text{ is } b\text{-edge } \big) \Rightarrow a \neq b$

In other words, 1) the leaves of $ST(S)$ represent positions of all suffixes of $S\$$, 2) concatenation of labels on the path from root to a leaf spells exactly the suffix represented by the leaf, 3) each internal node is a branching node - it has at least 2 children, 4) internal nodes represent common prefix of their descendants and 5) all labels of edges outgoing from a node must begin with distinct characters. Note that suffix tree definition uses suffixes that are extended to the right endmarker \$.

We define set $Prefixes(T) = \{prefix_T(v)|v \in V_I(T)\}$

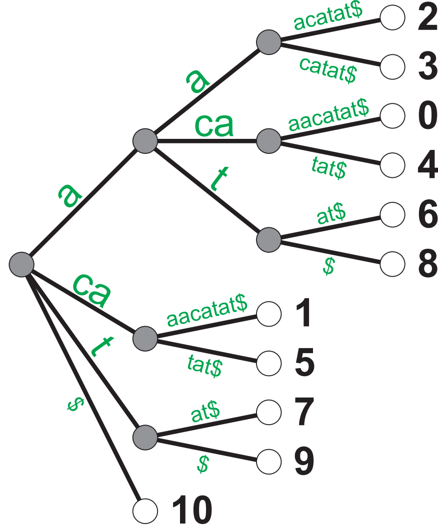

Figure 2.2: Suffix tree for string S='acaaacatat'

For an internal node v, we define $lcplen_T(v) = |prefix_T(v)|$.

Leaves of the suffix tree represent positions of suffixes of S. Internal nodes represent sets of positions. For an internal node v we define

$$Pos_T(v) = Children_T(v) \cap V_L(T)$$
$$Pos_T^+(v) = Desc_T(v) \cap V_L(T)$$

For a node $v \in V$

$$Pos_T^*(v) = \begin{cases} Pos_T^+(v) \text{ if } v \in V_I(T) \\ \{v\} \text{ if } v \in V_L(T) \end{cases}$$

Suffix tree for string S will be denoted $ST(S)$. Let $n = |S| + 1$ (length of $S\$$). Following properties should be easy to verify.

Property 2.1. $|V_I(T)| < n$.

Property 2.2. $\sum_{v \in V_I(T)} |Children_T(v)| < 2n$.

Property 2.3. $\{Pos_T(v) | v \in V_I(T)\}$ *is partition of* $V_I(T)$.

Property 2.4. $\forall v \in V_I(T) : Pos_T(v) \leq |\Sigma|$.

$map_T : N_{|S|+1} \to V_I(T)$ is a function that maps positions into internal nodes they belong to. $\forall i \in N_{|S|+1} : i \in Pos_T(map_T(i))$.

It is known that suffix tree T for string of length n can be built in $O(n)$ time and space, using algorithms of [3], [4] or [5] ([5] presents on-line construction). Function map_T can be realised

by table that can be easily precomputed in $O(n)$ time by one traversal of T. Value $map_T(i)$ can be therefore accessed in $O(1)$ time.

We deal with suffix trees, because they have an interresting property from the point of view of maximal repeats. Each internal node v of a suffix tree T represents the longest common prefix $prefix_T(v)$ of it's descendants. Moving to a child of the vertex v means extending the prefix. For suffixes under two distinct children (that are either leaf or internal nodes) the prefix $prefix_T(v)$ is not right-extensible.

Property 2.5. *Let v be an internal node of $ST(S)$, $l = lcplen(v)$, $c_1, c_2 \in Children(v)$, $c_1 \neq c_2$, $p_1 \in Pos^*(c_1)$, $p_2 \in Pos^*(c_2)$. Then (p_1, p_2, l) is right maximal repeat in S.*

Lemma 2.1. *Let $T = ST(S)$ and $p_1, p_2 \in N_{|S|+1}, p_1 \neq p_2$. Let $v_1 = map_T(p_1)$, $v_2 = map_T(p_2)$, $w = LCA_T(v_1, v_2)$. Then (p_1, p_2, l) is right maximal repeat in S if and only if $l = lcplen_T(w)$.*

Proof. It holds that $\exists c_1, c_2 \in Children(w)$, $p_1 \in Pos^*(c_1)$, $p_2 \in Pos^*(c_2)$. Also $c_1 \neq c_2$, because $w = LCA_T(v_1, v_2)$. By property 2.5 $(p_1, p_2, lcplen_T(w))$ is right maximal repeat in S. There can't be right maximal repeat (p_1, p_2, l) for l other than $lcplen_T(w)$, because it would contradict with properties of maximal repeat. □

Suffix tree will be used to define R3 Tree in the next chapter. However our implementation will be based on more space efficient structure - enhanced suffix array. Authors of article [2] show that suffix tree can be replaced by enhanced suffix array in many applications. They introduce conceptual data structure - lcp-interval tree that is defined by suffix array and lcp-table (it doesn't really have to be built). With little added information, this data structure allows us to simulate suffix tree traversals efficiently. In following text we'll introduce concept of enhanced suffix arrays and establish isomorphism of lcp-interval tree and suffix tree.

Suffix array is a permutation $sa_S : N_{|S|+1} \rightarrow N_{|S|+1}$ such that $(\forall i, j)(0 \leq i < j \leq |S|) : suffix_S(sa_S(i)) <_L suffix_S(sa_S(j))$. For $0 \leq i \leq j \leq |S|$ we define $sa_S[i..j] = \{sa_S(i), sa_S(i+1), \ldots, sa_S(j)\}$.

Lcp-table is a function $lcptab_S : N_{|S|+1} \rightarrow N_{|S|+1}$ such that $(\forall i)(1 \leq i \leq |S|) : lcptab_S(i) = lcplen\,(suffix_S(sa_S(i-1)), suffix_S(sa_S(i)))$, $lcptab_S(0) = 0$

Suffix array and lcp-table are represented as arrays of integers. It is known that suffix array for a string of length n can be constructed in $O(n)$ time and space. There are three different algorithms described in [6], [7] and [8].

Example of suffix array and lcp-table is shown on figure 2.3. Also values $LC_S(sa_S(i))$ and $suffix_S(sa_S(i))$ are shown for each suffix array entry $sa_S(i)$.

Figure 2.3: Suffix array and lcp-table for S='acaaacatat'.

i	$sa_S(i)$	$lcptab_S(i)$	$LC_S(sa_S(i))$	$suffix_S(sa_S(i))$
0	2	0	c	aaacatat$
1	3	2	a	aacatat$
2	0	1	c	acaaacatat$
3	4	3	a	acatat$
4	6	1	c	atat$
5	8	2	t	at$
6	1	0	a	caaacatat$
7	5	2	a	catat$
8	7	0	a	tat$
9	9	1	a	t$
10	10	0	t	$

Lcp-interval in table $lcptab_S$ is a triple (l, i, j) that satisfies all following conditions[1]

1) $0 \leq i < j \leq |S|$
2) $lcptab_S(i) < l$
3) $(\forall k)(i + 1 \leq k \leq j) : lcptab_S(k) \geq l$
4) $(\exists k)(i + 1 \leq k \leq j) : lcptab_S(k) = l$
5) $lcptab_S(j + 1) < l$

Sometimes, instead of triples, we'll use symbols $\boldsymbol{I}, \boldsymbol{J}, \ldots$ for lcp-intervals. For lcp-interval $\boldsymbol{I} = (l, i, j)$ we define $\boldsymbol{I}.lcp = l, \boldsymbol{I}.lb = i, \boldsymbol{I}.rb = j$.

Lcp-interval $\boldsymbol{I}$ is said to be **embedded** in lcp-interval $\boldsymbol{J}$ if $\boldsymbol{J}.lb \leq \boldsymbol{I}.lb \wedge \boldsymbol{I}.rb \leq \boldsymbol{J}.rb \wedge \boldsymbol{I}.lcp > \boldsymbol{J}.lcp$[2]. $\boldsymbol{J}$ is then called the interval enclosing $\boldsymbol{I}$. We call $\boldsymbol{I}$ a **child interval** of $\boldsymbol{J}$ if it is embedded in $\boldsymbol{J}$ and there is no other interval embedded in $\boldsymbol{J}$, that also encloses $\boldsymbol{I}$. We'll write $childint_S(\boldsymbol{J}, \boldsymbol{I})$ to express that $\boldsymbol{I}$ is child interval of $\boldsymbol{J}$.

$Lcpintervals(S)$ will denote set of all lcp-intervals in $lcptab_S$.

Lcp-interval tree for a string S is a tree $T = (V, E, root)$ such that

1) $V = Lcpintervals(S)$
2) $E = \{(\boldsymbol{I}, \boldsymbol{J}) \in V \times V | childint_S(\boldsymbol{I}, \boldsymbol{J})\}$
3) $root = (0, 0, |S|)$

[1]for the purpose of this definition we define $lcptab_S(|S| + 1) = -1$

[2]Note that we cannot have both $\boldsymbol{J}.lb = \boldsymbol{I}.lb$ and $\boldsymbol{I}.rb = \boldsymbol{J}.rb$ because $\boldsymbol{I}.lcp > \boldsymbol{J}.lcp$

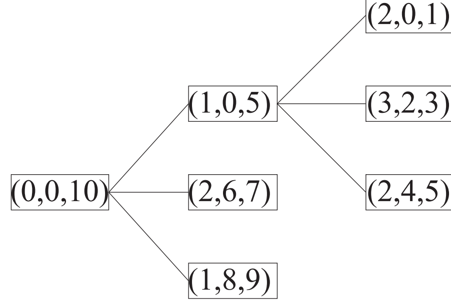

Figure 2.4: Lcp-interval tree for string S='acaaacatat'

$Pos^+_T(\boldsymbol{I}) = sa_S[\boldsymbol{I}.lb..\boldsymbol{I}.rb]$

$Pos_T(\boldsymbol{I}) = Pos^+_T(\boldsymbol{I}) \setminus \left(\bigcup_{\boldsymbol{J} \in Children_T(\boldsymbol{I})} Pos^+_T \boldsymbol{J} \right)$

For an lcp-interval $\boldsymbol{I} \in V$ we define $prefix_T(\boldsymbol{I})$ to denote the longest common prefix of all positions in $Pos^+_T(\boldsymbol{I})$.

$Prefixes(T) = \{prefix_T(\boldsymbol{I}) | \boldsymbol{I} \in V\}$

Each interval gets unique prefix and therefore we can define function $prefix^{-1}_T : Prefixes(T) \rightarrow V$ returning interval with given prefix.

Lcp-interval tree for a string S will be denoted $LCPIT(S)$.

Theorem 2.1. *Let* $T_1 = ST(S)$, $T_2 = LCPIT(S)$. *Then*

1) $Prefixes(T_1) = Prefixes(T_2)$

$h = prefix_{T_1} \circ prefix^{-1}_{T_2}$ *is isomorphism of* $T_I(T_1)$ *and* T_2, *i.e.*

2) h *is bijection between* $V_I(T_1)$ *and* $V(T_2)$

3) $(\forall (u, v) \in E_I(T_1)) : (h(u), h(v)) \in E(T_2)$

Internal node and it's respective lcp-interval have identical position sets.

4) $(\forall v \in V_I(T_1)) : Pos^+_{T_1}(v) = Pos^+_{T_2}(h(v))$

5) $(\forall v \in V_I(T_1)) : Pos_{T_1}(v) = Pos_{T_2}(h(v))$

This theorem establishes isomorphism of suffix tree (it's internal part) and lcp-interval tree. From now on, we don't have to distinguish between internal nodes of suffix tree and lcp-intervals, $V(LCPIT(S)) = V_I(ST(S))$. This theorem is presented without proof but we believe it should be clear from properties of lcp-intervals and lcp-interval tree. Detailed proof would be out of scope of this thesis. For details refer to [2].

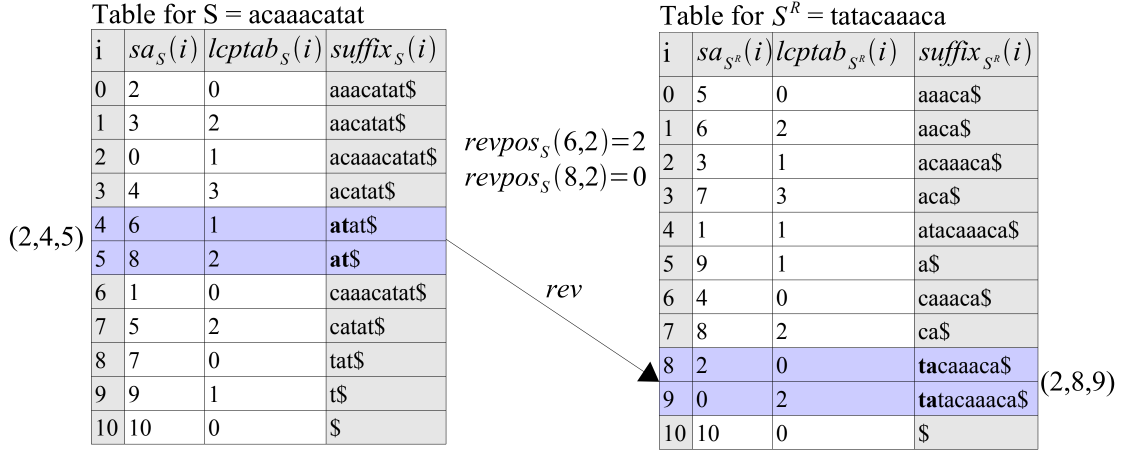

Table for S = acaaacatat

i	$sa_S(i)$	$lcptab_S(i)$	$suffix_S(i)$
0	2	0	aaacatat$
1	3	2	aacatat$
2	0	1	acaaacatat$
3	4	3	acatat$
4	6	1	**at**at$
5	8	2	**at**$
6	1	0	caaacatat$
7	5	2	catat$
8	7	0	tat$
9	9	1	t$
10	10	0	$

Table for S^R = tatacaaaca

i	$sa_{S^R}(i)$	$lcptab_{S^R}(i)$	$suffix_{S^R}(i)$
0	5	0	aaaca$
1	6	2	aaca$
2	3	1	acaaaca$
3	7	3	aca$
4	1	1	atacaaaca$
5	9	1	a$
6	4	0	caaaca$
7	8	2	ca$
8	2	0	**ta**caaaca$
9	0	2	**ta**tacaaaca$
10	10	0	$

Figure 2.5: Example of reverse lcp-interval in suffix array and lcp-table.

2.3 Reverse lcp-interval

Authors of suffix tree construction algorithms ([5], [3]) work with concept of auxiliary edges called suffix links. Usually, when taking edges in the suffix tree, the prefix represented by the current node is lengthened or shortened at the end. Suffix links are used to move from one node to another so that the represented prefix is shortened at the front. In [11] it is shown that, there is a strong relationship between the suffix tree built from the reverse string (often called reverse prefix tree) and the suffix links of the suffix tree built from the original string. Utilization of this relationship of $ST(S^R)$ and $ST(S)$ is brought one step further by introduction of Affix trees in [12]. In our work, we exploit some aspects of this relationship with the concept of reverse lcp-intervals.

Let $x = S[p..p+l-1]$ be a substring of length l, that occurs in S at position p. It holds

$$(S[p..p+l-1])^R = S^R[|S|-(p+l)..|S|-p-1]$$

To express position of x^R in S^R, we define function

$$revpos_S(p,l) = |S| - (p+l)$$

If P is set of positions, then

$$revpos_S(P,l) = \{revpos_S(p,l) | p \in P\}$$

Theorem 2.2. *Let $T = LCPIT(S)$, $T^R = LCPIT(S^R)$, $\boldsymbol{I} \in V(T)$. Then there exists exactly one $\boldsymbol{J} \in V(T^R)$ such that*

1) $\boldsymbol{J}.lcp \geq \boldsymbol{I}.lcp$

2) $prefix_T(\boldsymbol{I})^R \sqsubseteq prefix_{T^R}(\boldsymbol{J})$

3) $Pos^+_{T_R}(\boldsymbol{J}) = revpos_S(Pos^+_T(\boldsymbol{I}), \boldsymbol{I}.lcp)$

We call the lcp-interval $\boldsymbol{J}$ a ***reverse lcp-interval*** *of $\boldsymbol{I}$ and we denote it $rev(\boldsymbol{I})$.*

Proof. Let $\alpha = prefix_T(\boldsymbol{I})$, let $(l, i, j) = \boldsymbol{I}$. From properties of suffix array it follows that $sa_S[i..j]$ is set of <u>all</u> positions at which α occurs in S and therefore $revpos_S(sa_S[i..j], l)$ is set of <u>all</u> positions at which α^R occurs in S^R. Let's have set of all suffixes of S^R, with prefix α^R

$$R = \{suffix_{S^R}(k) | k \in revpos_S(sa_S[i..j], l)\}$$

Let p be such that $suffix_{S^R}(sa_{S^R}(p))$ is minimum from R w.r.t. $<_L$.
Let q be such that $suffix_{S^R}(sa_{S^R}(q))$ is maximum from R w.r.t. $<_L$.

Since sa_{S^R} is lexicographically ordered, we have $sa_{S_R}[p..q] \supseteq revpos_S(sa_S[i..j], l)$. $suffix_S$ and $revpos_S$ are bijections, therefore $sa_{S^R}[p..q] = revpos_S(sa_S[i..j], l)$ and $R = \{suffix_{S^R}(k) | k \in sa_{S^R}[p..q]\}$.

Let β be the longest common prefix of all suffixes from R. and let $m = |\beta|$. It's easy to see that α^R is prefix of β and therefore $m \geq l$. Now we will show that $\boldsymbol{J} = (m, p, q)$ is lcp-interval in $lcptab_{S^R}$, i.e. it satisfies all conditions

1) $0 \leq p < q \leq |S^R|$

2) $lcptab_{S^R}(p) < m$

3) $(\forall k)(p + 1 \leq k \leq q) : lcptab_{S^R}(k) \geq m$

4) $(\exists k)(p + 1 \leq k \leq q) : lcptab_{S^R}(k) = m$

5) $lcptab_{S^R}(q + 1) < m$

1) $p < q$ because $|R| \leq 2$.
2) $lcptab_{S^R}(p) < l$ because p is the lowest index in sa_{S^R} such that $suffix_{S^R}(sa_{S^R}(p))$ begins with α^R. $l \leq m \Rightarrow lcptab_{S^R}(p) < m$.
3) and 4) follow from the choice of m.
5) $lcptab_{S^R}(q + 1) < l$ because q is the highest index in sa_{S^R} such that $suffix_{S^R}(sa_{S^R}(q))$ begins with α^R. $l \leq m \Rightarrow lcptab_{S^R}(q + 1) < m$.

From this we have $\boldsymbol{J} = rev(\boldsymbol{I})$. □

Lemma 2.2. *Let $T = LCPIT(S)$, $T^R = LCPIT(S^R)$, $V_1 = \{\boldsymbol{I} \in V(T) | rev(\boldsymbol{I}).lcp = \boldsymbol{I}.lcp\}$. Then rev narrowed to V_1 is injection i.e. $\boldsymbol{I}, \boldsymbol{J} \in V_1 \wedge \boldsymbol{I} \neq \boldsymbol{J} \Rightarrow rev(\boldsymbol{I}) \neq rev(\boldsymbol{J})$.*

Proof. Let $\boldsymbol{R} = rev(\boldsymbol{I}) = rev(\boldsymbol{J})$. It holds that
$Pos^{+}_{T^R}(\boldsymbol{R}) = revpos_S(Pos^{+}_{T}(\boldsymbol{I}), \boldsymbol{I}.lcp) = revpos_S(Pos^{+}_{T}(\boldsymbol{J}), \boldsymbol{J}.lcp)$
As $\boldsymbol{I}.lcp = \boldsymbol{J}.lcp = \boldsymbol{R}.lcp$, we can also write
$revpos_{S^R}(Pos^{+}_{T^R}(\boldsymbol{R}), \boldsymbol{R}.lcp) = Pos^{+}_{T}(\boldsymbol{I}) = Pos^{+}_{T}(\boldsymbol{J})$

and from that we have $\boldsymbol{I} = \boldsymbol{J}$. □

Chapter 3

R3 Tree

Property 2.5 of suffix trees would be sufficient for us to be able to report all *right maximal* repeats. To ensure left maximality of (l, i, j), it has to hold that $LC_S(i) \neq LC_S(j)$. In following section we'll define another refinement of the structure of the suffix tree, so that we can address this requirement. This will be done by partitioning the Pos_T sets of internal nodes of suffix tree T according to left contexts of its suffixes/positions. The result will be a conceptual structure - LC-bucket tree - that will be used to illustrate problems we need to cope with, when we want to implement optimal findPairs query.

3.1 LC-bucket tree

Let $\Sigma_{\cent} = \Sigma \cup \{\cent\}$. For an internal node v of suffix tree T for a string S and symbol a we define **LC-bucket** to be a set of positions returned by function $b : V_I(T) \times \Sigma_{\cent} \to 2^{N_{|S|+1}}$ which is defined as follows:

$$b_T(v, a) = \{i \in Pos_T(v) | LC_S(i) = a\}$$

We also define analogue for sets Pos^+_T that will be used later. We will call it b^+ set.

$$b^+_T(v, a) = \{i \in Pos^+_T(v) | LC_S(i) = a\}$$

Set of all LC-buckets for a suffix tree T:

$$B(T) = \{b_T(v, a) | v \in V_I(T), a \in \Sigma_{\cent}, b_T(v, a) \neq \emptyset\}$$
$$B^+(T) = \{b^+_T(v, a) | v \in V_I(T), a \in \Sigma_{\cent}, b^+_T(v, a) \neq \emptyset\}$$

We define left context for a b^+ set to be left context of any of it's elements (there has to be at least one and there is finite number of them).

$$\forall B \in B^+(T) : LC_S(B) = LC_S(i) \text{ for some } i \in B$$

For better legibility of text, we omit parameter T in expressions $b_T(v,a)$ and $b^+_T(v,a)$. When we use symbols $b(v,a)$ and $b^+(v,a)$ we allways know the suffix tree T that node v belongs to.

These are easily verifiable facts about buckets and b^+ sets. We present them without proof.

Lemma 3.1. *Let $T = ST(S)$, $T_I = T_I(T)$, $v \in V_I(T)$, $a \in \Sigma_{\mathbb{c}}$.*

1) $B(T)$ *is partition of* $N_{|S|+1}$.

2) $b^+(v,a) = b(v,a) \cup \left(\bigcup_{u \in Children_{T_I}(v)} b^+(u,a) \right)$

3) $u_1, u_2 \in Children_{T_I}(v) \wedge u_1 \neq u_2 \Rightarrow b^+(u_1,a) \cap b^+(u_2,a) = \emptyset$

4) $u \in Children_{T_I}(v) \Rightarrow b^+(u,a) \cap b(v,a) = \emptyset$

Lemma 3.2. *Let $T = ST(S)$, $T_I = T_I(T)$, $E_I = E_I(T)$, $u, v \in V_I(T)$, $a \in \Sigma_{\mathbb{c}}$.*

1) $b^+(u,a) = b^+(v,a) \neq \emptyset \Rightarrow \big((u,v) \in E_I^+ \vee (v,u) \in E_I^+ \vee u = v\big)$

2) $\big(u \neq v \wedge (u,v) \notin E_I^+ \wedge (v,u) \notin E_I^+\big) \Rightarrow b^+(u,a) \cap b^+(v,a) = \emptyset$

3) $b^+(u,a) \supset b^+(v,a) \neq \emptyset \Rightarrow (u,v) \in E_I^+$

Proof. 1) Let's consider case $u \neq v$. Let $s \in b^+(u,a)$. Let $s = u_0, u_1, u_2, \ldots, u_k = u$ be path from s to u in T and let $s = v_0, v_1, v_2, \ldots, v_l = v$ be path from s to v in T. W.L.O.G., let $0 < k < l$. Then $u_0 = v_0 = s$, $u_1 = v_1 = parent_T(s)$, $\ldots$, $u_k = v_k = parent_T(u_{k-1})$, $u = v_k, v_{k+1}, \ldots, v_l = v$ is path in T, therefore $(v,u) \in E_I^+$.

2) Let $w = LCA_T(u,v)$. $\big(u \neq v \wedge (u,v) \notin E_I^+ \wedge (v,u) \notin E_I^+\big) \Rightarrow (w \neq u \wedge w \neq v)$. $(\exists w_1, w_2 \in Children_{T_I}(w))(w_1 \neq w_2)((w_1,u) \in E_I^* \wedge (w_2,v) \in E_I^*)$, where E_I^* is reflexive and transitive closure of E_I. By lemma 3.1 3) $b^+(w_1,a) \cap b^+(w_2,a) = \emptyset \Rightarrow b^+(u,a) \cap b^+(v,a) = \emptyset$.

3) $b^+(u,a) \supset b^+(v,a) \neq \emptyset \Rightarrow b^+(u,a) \cap b^+(v,a) \neq \emptyset \Rightarrow$ by 2) $u = v \vee (u,v) \in E_I^+ \vee (v,u) \in E_I^+$. Neither $(v,u) \in E_I^+$ nor $u = v$ can be the case because it would imply $b^+(u,a) \subseteq b^+(v,a)$, which would contradict our premise. Therefore $(u,v) \in E_I^+$ holds.

□

Let's have $T = ST(S)$ and $M \in B^+(T)$, $a = LC_S(M)$. There may be more than one $v \in V_I(T)$ such that $b^+(v,a) = M$. However by statement 1) of lemma 3.2, all such nodes v form path in T. Therefore we can define functions returning first and last(deepest) node of this path.

$$first_T(M) = v \in V_I(T) : b^+(v,a) = M \wedge b^+(parent_T(v),a) \neq M$$
$$last_T(M) = v \in V_I(T) : b^+(v,a) = M \wedge (\forall u \in Children_(T_I(T)) : b^+(v,a) \neq M)$$

LC-bucket tree will be created from suffix tree by removing it's leaves and appending LC-buckets to respective internal nodes.

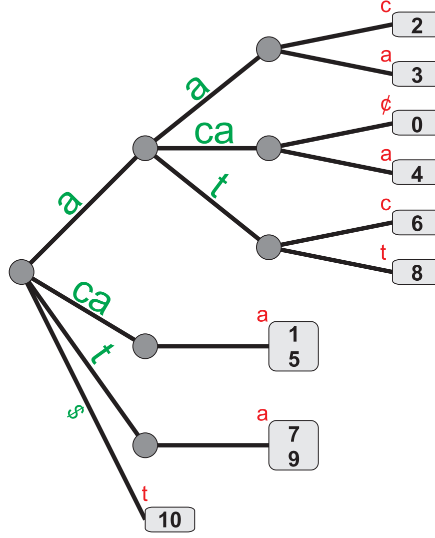

Figure 3.1: LC-bucket tree for string S='acaaacatat'

LC-bucket tree for string S is a tree $T = (V, E, root)$, such that

1) $V = V_I(ST(S)) \cup B(ST(S))$
2) $E = E_I(ST(S)) \cup E_B$
3) $root = root(ST(S))$

Where $E_B = \{(v, b(v, a)) | v \in V_I(ST(S)), a \in \Sigma_{¢}, b(v, a) \neq \emptyset\}$

By removing leaves and adding LC-buckets, no internal node of ST(S) could become leaf. Also no bucket node has any children, therefore $V_I(T) = V_I(ST(S))$, $V_L(T) = B(T)$, $E_I(T) = E_I(ST(S))$ and $E_L(T) = E_B$.

We define lcplen function for internal nodes of T:

$\forall v \in V_I(T) : lcplen_T(v) = lcplen_{ST(S)}(v)$

LC-bucket tree for a string S will be denoted $LCBT(S)$.

3.2 Non-optimal findPairs query on LC-bucket tree

We present LC-bucket tree and non-optimal version of findPairs query as the first iteration towards our target data structure, to demonstrate the main problems that have to be solved to achieve optimality. The algorithm is given on figure 3.2. It works with LC-bucket tree $T = LCBT(S)$. Values $map_T(i)$, $LC_S(i)$, $lcplen_T(v)$, $parent_T(v)$ are assumed to be accessible in $O(1)$ time. Also all children of a node can be determined in constant time.

Theorem 3.1. *Call of function findPairsNonOptimal(p_1, k) reports pair (p_2, l) if and only if (p_1, p_2, l) is maximal repeat in S such that $l \geq k$.*

Proof. Let $T = LCBT(S)$, $v_1 = map_T(p_1)$, $v_2 = map_T(p_2)$, and $w = LCA_T(v_1, v_2)$. Let $u_1 = v_1$, $u_2 = parent_T(u_1)$, $\ldots$, $u_p = parent_T(u_{p-1})$ be the sequence of nodes on path from v_1 to $root(T)$ visited by procedure findPairsNonOptimal in the while loop on lines 3-6. u_p is the last node with $lcplen_T(u_p) \geq k$. Note that once a subtree of a node u_j is marked as visited in line 5 it is never visited again in any subsequent calls of combineSubtree on line 4. This is because next node to be marked is it's parent u_{j+1}, whose marking also excludes subtree of u_j.

(if) Let (p_1, p_2, l) be a maximal repeat in S such that $l \geq k$. Since (p_1, p_2, l) is right maximal, by lemma 2.1 $lcplen_T(w) = l \geq k$ and therefore $\exists j \in \{1..p\} : w = u_j$. It means that procedure combineSubtree is called at least once with parameters $pos := p_1$, $v := w$, $visited := u_{j-1}$ (where $u_0 = \perp$), $l := lcplen_T(w)$. Procedure combineSubtree recursively visits all nodes in subtree of w that weren't already visited by previous call from findPairsNonOptimal and for all buckets B with $LC_S(B) \neq LC_S(p_1)$ reports pair (s, l) for each $s \in B$. Since w is ancestor of v_2, node v_2 is also visited by combineSubtree and since w is lowest common ancestor of v_1 and v_2, v_2 hasn't been visited in any previous call. Since (p_1, p_2, l) is left maximal $LC_S(p_1) \neq LC_S(p_2)$ and therefore (p_2, l) is also reported.

(only if) Let (p_2, l) be a pair reported by a call findPairsNonOptimal(p_1, k). It had to be reported by a call of procedure combineSubtree on u_j for some j in line 4 of findPairsNonOptimal. This means that $l \geq k$. The pair (p_2, l) could only be reported if $LC_S(p_1) \neq LC_S(p_2)$, because of condition in line 3 of combineSubtree. u_j is common ancestor of v_1 and v_2. We will show that it is also lowest common ancestor w. Suppose that $w = u_i$ for some i such that $depth_T(u_i) > depth_T(u_j)$. This means that $i < j$ and that (p_2, l) is reported by a call combineSubtree on the node u_i which is then marked as visited on line 5 before combineSubtree on u_j is called. This means however that combineSubtree called on node u_j couldn't report pair (p_2, l), which is contradiction. Since $u_j = w$, it holds $l = lcplen_T(w)$ and since $w = LCA_T(v_1, v_2)$, by

```
combineSubtree(pos, v, visited, l)
1    for all children c of v
2        if c ≠ visited
3          if c is bucket and LC_S(c) ≠ LC_S(pos)
4            for all s from c
5                report pair (s, l)
6          if c is iternal node
7            combineSubtree(pos, c, visited, l)

findPairsNonOptimal(pos,k)
1    v := map_T(pos)
2    visited := ⊥
3    while v ≠⊥ and lcplen_T(v) ≥ k do
4        combineSubtree(pos, v, visited, lcplen_T(v))
5        visited := v
6        v := parent_T(v)
```

Figure 3.2: Algorithm 1 - Non-optimal findPairs query on LC-bucket tree

lemma 2.1 we have (p_1, p_2, l) is right maximal repeat. Therefore (p_1, p_2, l) is maximal repeat in S with $l \geq k$.

□

The problem is, that findPairsNonOptimal may take $O(n)$ time, while only 1 pair is reported. Let's take string $S = a^n$ for example. $T = LCBT(a^n)$ has n internal nodes that form a single path. Each of them, except the deepest one, has one LC-bucket. All of suffixes have left context a except the suffix 0 which has left context ¢. Let u_i be node with depth i. $lcplen_T(u_i) = i$. LC-bucket of each u_i contains exactly one suffix $n - i$. The deepest node u_{n-1} contains also LC-bucket with suffix 0. If we call findPairsNonOptimal($n - i$, 1) for $n - i \neq 0$, we start while loop at lines 3-6 with node $u_i = map_T(n - i)$ and end at node u_1. First call of combineSubtree will traverse $n - i + 1$ buckets under u_i. Subsequent calls of combineSubtree on nodes $u_{i-1}, u_{i-2}, \ldots, u_1$ will traverse one bucket on each. This means another $i - 1$ buckets. Only one of these n buckets has left context other than a (the node u_{n-1}) and therefore only one pair is reported.

The non-optimality of algorithm findPairsNonOptimal comes from following two problems

1) combineSubtree(p_1, v, $visited$, l) visits all buckets under node v, not only buckets with left context other than $LC_S(p_1)$. This makes time consumed by combineSubtree not proportional to number of reported pairs.

2) while loop on lines 3-6 of findPairsNonOptimal visits all nodes v with $lcplen_T(v) \geq k$ on the path from $map_T(p)$ to root, disregarding that some nodes may not contain any unvisited bucket with left context other than $LC_S(p)$. This means, that we might visit too many nodes without proportional number of pairs being reported.

The R3 tree structure presented in the following text solves exactly these two problems.

3.3 Union trees

First problem of non-optimal findPairs query described in previous section is that it visits all buckets in the subtree of the LC-bucket tree node v on which it is called and not just the buckets with left context different from $LC_S(p_1)$. Positions we want to report are only those from the sets $b^+(v, a)$ where $a \neq LC_S(p_1)$. It would be useful, if we could access b^+ sets for certain left contexts. If we want to do this, we have to represent them in a data structure that allows optimal access to their contents and also has optimal memory requirements. In this section we describe the way, how to represent the sets from $B^+(T)$. Unlike LC-buckets, b^+ sets for different suffix tree nodes may overlap. We'll have to apply few tricks so that we don't have to store the same position in two different places. Elements of $B^u(T) = B^+(T) \setminus B(T)$ will be called **union nodes**, because they can be constructed by union of buckets from $B(T)$:

$$\forall B \in B^u(T) : \exists B_1, B_2, \ldots, B_k \in B(T) : B = B_1 \cup B_2 \cup \ldots \cup B_k$$

Our next conceptual structure - union tree - captures the structure of union operators applied to LC-buckets to be composed into union node.

Union tree for a suffix tree $T = ST(S)$ and $B \in B^+(T)$, is a tree $UT_T(B) = (V, E, root)$ such that

1) $V = \{b^+(u,a) | (v,u) \in E_I^*, b^+(u,a) \neq \emptyset\}$, where E_I^* is reflexive and transitive closure of $E_I(T)$, $v = last_T(B)$ and $a = LC_S(B)$.

2) $E = \{(M_1, M_2) \in V \times V | M_1 \supset M_2 \wedge \neg(\exists M_3 \in V : M_1 \supset M_3 \supset M_2)\}$

3) $root = B$

We will use symbol UT_T without parameter T for simplicity. For a suffix tree T, $UF(T)$ will denote forest of union trees $UT(b^+(root(T), a))$ for all a. Symbol $emptyUT = UT(\emptyset)$ will denote special empty union tree.

Lemma 3.3. *Let $T = ST(S)$, $v \in V_I(T)$, $a \in \Sigma_{\mathfrak{c}}$, $b(v,a) \neq \emptyset$, $T_u = UT(b^+(v,a))$. Then either $b^+(v,a) = b(v,a)$ or $(b^+(v,a), b(v,a)) \in E(T_u)$.*

Proof. Let $b^+(v,a) \supset b(v,a)$ and $(b^+(v,a), b(v,a)) \notin E(T_u)$ i.e. $(\exists M_3 \in V(T_u))(b^+(v,a) \supset M_3 \supset b(v,a))\}$. Let u be child of v such that $b^+(u,a) \supseteq M_3$ (3) in lemma 3.2 guarantees it's existence). From this we have $b^+(u,a) \cap b(v,a)$, which contradicts 3) in lemma 3.1. □

Following property is easily verifiable.

Property 3.1. *Let u, v be internal nodes of $ST(S)$ such that u is descendant of v, $a \in \Sigma_{\mathfrak{c}}$. Then $UT(b^+(v,a)) = UT(b^+(u,a))$ or $UT(b^+(u,a))$ is subtree of $UT(b^+(v,a))$*

Lemma 3.4. *Let $T = ST(S)$, $T_I = T_I(T)$, $U \in B^u(T)$ $a = LC_S(U)$, $T_u = UT(U)$. Then $|Children_{T_u}(U)| \geq 2$.*

Proof. Let $v = last_T(U)$. Let $c_1, c_2, \ldots, c_k$ be all elements of $Children_{T_I}(v)$.

$(\forall i)(1 \leq i \leq k) : b^+(c_1, a) \neq U$. (1)

By 2) in lemma 3.1 $U = b^+(v,a) = b(v,a) \cup b^+(c_1,a) \cup \ldots \cup b^+(c_k,a)$. Sets $b(v,a), b^+(c_1,a), \ldots, b^+(c_k,a)$ are disjunct by 3) and 4) of lemma 3.1. One of them has to be non-empty. Let's say that there is exactly one non-empty set among them.

- If it is $b(v,a)$ then $U = b(v,a)$ which contradicts that U is an union node.
- If it is $b^+(c_i,a)$ for an i, $1 \leq i \leq k$ then it's contradiction with (1)

So there have to be two or more non-empty sets among $b(v,a), b^+(c_1,a), \ldots, b^+(c_k,a)$.

- If $b(v,a)$ is among them, by lemma 3.3 $(U, b(v,a)) \in E(T_u)$
- If $b^+(c_i,a)$ is among them, then $(U, b^+(c_i,a)) \in E(T_u)$, if the edge $(U, b^+(c_i,a))$ didn't exist it would mean that $(\exists b^+(x,a) \in V(T_u))(b^+(v,a) \supset b^+(x,a) \supset b^+(c_i,a))\} \Rightarrow$ by 3) in lemma 3.2 $(v,x) \in E_I^+(T) \wedge (x,c_i) \in E_I^+(T) \Rightarrow c_i \notin Children_{T_I}(v)$ which is contradiction.

□

For an union tree T_u, $n_u(T_u)$ denotes number of it's union nodes and $n_b(T_u)$ number of it's bucket nodes. $n_u(emptyUT) = n_b(emptyUT) = 0$.

Lemma 3.5. *Let $T = ST(S)$, $B \in B^+(T)$, $T_u = UT(B)$. Then $n_u(T_u) < n_b(T_u)$.*

Proof. We will prove the lemma by induction on $n_u(T_u)$. If $n_u(T_u) = 0$, T_u consists of single bucket and $n_u(T_u) < n_b(T_u)$ holds. Let $n_u(T_u) = k$ and suppose the statement $n_u(T_1) < n_b(T_1)$ holds for all union trees T_1 with $n_u(T_1) < k$.

$k > 0 \Rightarrow B$ is union node and by lemma 3.4 $|Children_{T_u}(B)| \geq 2$

Let $C_1, C_2, ..., C_n$ be all children of B.

$n_b(T_u) = \sum_{1 \leq i \leq n} n_b(UT(C_i))$
$n_u(T_u) = 1 + \sum_{1 \leq i \leq n} n_u(UT(C_i))$
$(\forall i)(1 \leq i \leq n)(n_u(UT(C_i)) < n_b(UT(C_i)))$

$n \geq 2 \Rightarrow n_u(T_u) < n_b(T_u)$

□

Lemma 3.6. *Let $T = ST(S)$, $B_1, B_2 \in B^+(T)$, $T_u = UT(B_1)$, $B_2 \in V(T_u)$. If T'_u is a tree, that results from prunning subtree $UT(B_2)$ from T_u then $n_u(T'_u) \leq n_b(T'_u)$.*

Proof. We will prove the lemma by induction on $n_u(T_u)$. For $n_u(T_u) = 0$, T_u is $emptyUT$ or single bucket and the goal holds. Let $n_u(T_u) = k$ and suppose the goal statement holds for all union trees T_1 with $n_u(T_1) < k$

$k > 0 \Rightarrow B_1$ is union node and by lemma 3.4 $|Children_{T_u}(B_1)| \geq 2$.

Let $C_1, C_2, ..., C_n$ be all children of B_1

There has to be j, $1 \leq j \leq n$ such that $B_2 \in V(UT(C_j))$ Let T_j be union tree that is created by prunning $UT(B_2)$ from $UT(C_j)$.

$n_b(T'_u) = n_b(T_j) + \sum_{1 \leq i \leq n, i \neq j} n_b(UT(C_i))$
$n_u(T'_u) = 1 + n_u(T_j) + \sum_{1 \leq i \leq n, i \neq j} n_u(UT(C_i))$
$(\forall i)(1 \leq i \leq n, i \neq j)(n_u(UT(C_i)) < n_b(UT(C_i)))$
$n_u(T_j) \leq n_b(T_j)$

$n \geq 2 \Rightarrow n_u(T'_u) \leq n_b(T'_u)$

□

We can conclude, that we are able to store $B^+(T)$ in $O(n)$ space ($T = ST(S)$, $n = |S| + 1$). Elements of $B(T)$ can be stored explicitly as they are. By 1) of lemma 3.1 $B(T)$ is partition

of N_n. We need $O(|B|)$ space to represent bucket $B \in B(T)$ and $|B(T)| \leq n$, therefore we will be able to represent $B(T)$ in $O(n)$ space.

Elements of $B^u(T)$ will be represented as union nodes. An union node needs only constant amount of information to be stored in form of pointers to it's children. By 2) of lemma 3.1, maximal number of such pointers is $|\Sigma| + 1$.

$B^+(T)$ will be stored in form of union forest $UF(T)$. Lemma 3.5 shows that for each $T_a = UT(b^+(root(T), a)) : n_u(T_a) < n_b(T_a)$ and therefore $|B^u(T)| < |B(T)|$. This way b^+ sets can be effectively represented in $O(n)$ space. Moreover, enumeration of all elements of $B \in B^+(T)$ can be done in $O(|B|)$ time.

This representation of sets in union trees also allows union operation in $O(1)$ time which will prove useful later on.

3.4 R3 tree

Let $T_S = ST(S)$. **R3 tree** for a string S is a 6-tuple $T = (V, E, root, UF, bp, up)$ such that

1) $V = V_I(T_S)$
2) $E = E_I(T_S)$
3) $root = root(T_S)$
4) $UF = UF(T_S)$
5) $bp : V \times \Sigma_{\mathfrak{c}} \rightarrow V(UF), bp(v, a) = b^+(v, a)$
6) $up : V \times \Sigma_{\mathfrak{c}} \rightarrow V$

R3 tree is internal part of suffix tree with few enhancements. Set of edges E is present so that we can speak of parent-child relationships of R3 tree nodes, but they don´t need to be represented as we will see later. For navigation in R3 tree structure, we will use two functions bp and up instead. The function bp connects each node with it's b^+ sets in union forest and up is a navigation function.

$up(v, a)$ is nearest ancestor u of v (in T_S), such that $Pos^+_{T_S}(u)$ contains at least one more position i with $LC_S(i) \neq a$ than $Pos^+_{T_S}(v)$.

Now we define condition that has to be met by node $u = up(v, a)$ more formally:

$$UP(u, v, a) \equiv (u, v) \in E^+ \wedge (\exists b \in \Sigma_{\mathfrak{c}})(b \neq a)(b^+(u, b) \supset b^+(v, b))$$

We also want $u = up(v, a)$ to be deepest with respect to $depth = depth_{T_I(T_S)}$:

$$DUP(u, v, a) \equiv UP(u, v, a) \wedge ((\exists w : UP(w, v, a) \wedge w \neq u) \Rightarrow depth(w) < depth(u))$$

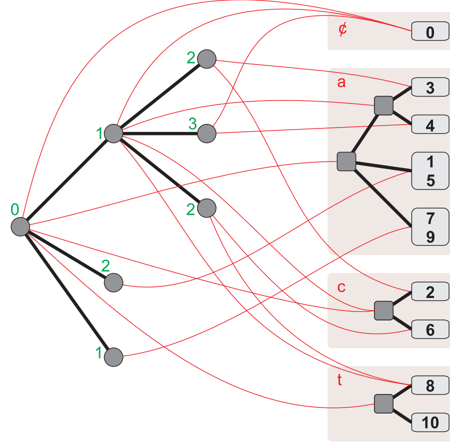

Figure 3.3: Visualisation of R3 tree for string S='acaaacatat' with union forest.

$$up(v, a) = \begin{cases} u \text{ such that } DUP(u, v, a) \text{ if such } u \text{ exists.} \\ \perp \text{ otherwise} \end{cases}$$

R3 tree for string S will be denoted $R3T(S)$.

We define $V(T) = V$, $E(T) = E$, $root(T) = root$.

Nodes of R3 tree are also internal nodes of a suffix tree or nodes of a lcp-interval tree. It holds $V(R3T(S)) = V_I(ST(S)) = V(LCPIT(S))$.

$Chidren_T = Children_{T_I(T_S)}$, $Desc_T = Desc_{T_I(T_S)}$, $parent_T = parent_{T_I(T_S)}$, $depth_T = depth_{T_I(T_S)}$, $LCA_T = LCA_{T_I(T_S)}$. $lcplen_T = (lcplen_{T_S}$ narrowed to $V_I(T_S))$, $map_T = map_{T_S}$.

For later analysis, we define following values for nodes of R3 tree.
$bpsize_T(v)$ is number of different symbols a, such that $bp(v, a) \neq \emptyset$
$upsize_T(v)$ is number of different symbols a, such that $up(v, a) \neq \perp$

bp and up can be seen as tables that are stored in each node v of R3 tree. We will call them bp-table and up-table for node v. $bpsize(v)$ resp. $upsize(v)$ is size of bp-table resp. up-table for node v. Important property of these tables is that bp and up values can be accessed in $O(1)$ time.

Figure 3.3 shows simple R3 tree with bp-table pointers displayed as links to nodes of union trees. Number next to each R3 tree node represents lcplen value. Another visualisation of R3 tree can be seen at figure 3.4. This time bp-tables and up-tables are shown for each node. Each R3 tree node and union tree node is labeled by a number to be reffered to in up-tables and bp-tables. Edges from E are hidden. Edges between R3 tree nodes represent values from up-tables.

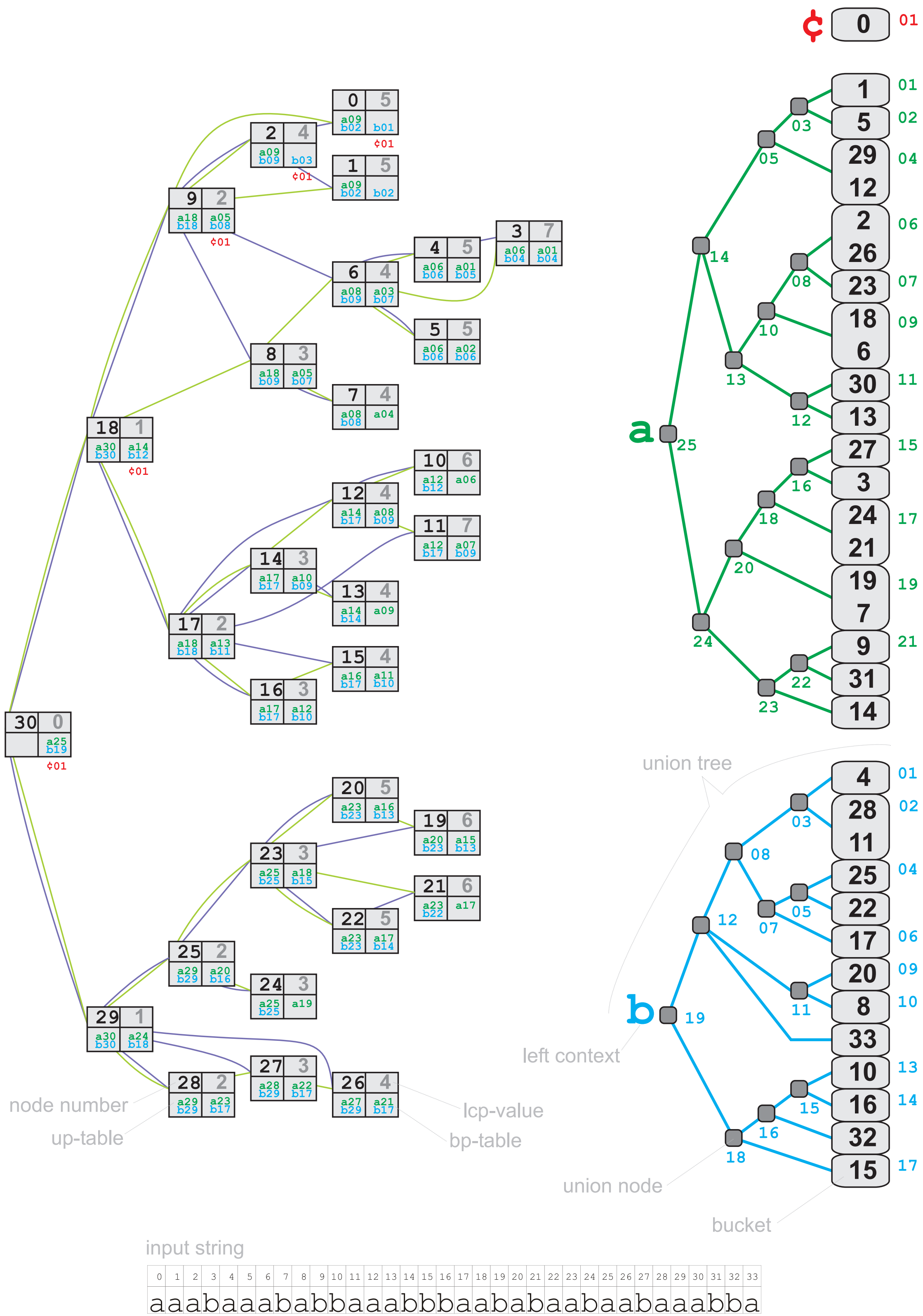

Figure 3.4: Another visualisation of R3 tree with union forest.

3.5 Optimal findPairs query on R3 tree

Optimal find pairs query is shown as Algorithm 2 on figure 3.5. It assumes we have string S and $T = R3T(S)$ available and also map_T is precomputed.

combineUnionSubtree(u, $visited$, l)
1 if $u = visited$ exit
2 for all children c of u
3 if c is bucket node
4 for all s from c
5 report pair (s, l)
6 if c is union node
7 combineUnionSubtree(c, $visited$, l)

findPairs(pos, k)
1 $v := map_T(pos)$
2 for each symbol a
3 $visited[a] := \perp$
4 while $v \neq \perp$ and $lcplen_T(v) \geq k$ do
5 for all a such that $bp(v, a) \neq \perp$
6 if $a \neq LC_S(pos)$
7 combineUnionSubtree($bp(v, a)$, $visited[a]$, $lcplen_T(v)$)
8 $visited[a] := bp(v, a)$
9 $v := up(v, LC_S(pos))$

Figure 3.5: Algorithm 2 - Optimal findPairs query on R3 tree

Lemma 3.7. *Call combineUnionSubtree(u, visited, l) runs in time $O(z)$, where z is number of reported pairs.*

Proof. Let $T = UT(u)$ and T' is tree that results from prunning $UT(visited)$ from T. Call combineUnionSubtree(u, $visited$, l) traverses T' and reports at least one pair for each bucket it encounters. Therefore $n_b(T') \leq z$. By lemma 3.6 $n_u(T') \leq n_b(T')$. Processing each node takes constant time, thus total time is $O(z)$. □

Theorem 3.2. *Call findPairs(p_1, k) reports pair (p_2, k) if and only if (p_1, p_2, l) is maximal repeat in S such that $l \geq k$.*

Proof. Let $T = R3T(S)$, $v_1 = map_T(p_1)$, $v_2 = map_T(p_2)$, $a_1 = LC_S(p_1)$, $a_2 = LC_S(p_2)$ and $w = LCA_T(v_1, v_2)$. Let $w_1 = v_1$, $w_2 = parent_T(w_1)$, $\ldots$, $w_s = parent_T(w_{s-1})$ be the sequence of nodes on path from v_1 to $root(T)$ such that $\forall i \in \{1..s\} : lcplen_T(w_i) \geq k$. $lcplen_T(parent_T(w_s)) < k$ or $w_s = root(T)$. This sequence may be empty if $lcplen_T(v_1) < k$. Let $u_1 = v_1$, $u_2 = up(u_1, a_1)$, $\ldots$, $u_p = up(u_{p-1}, a_1)$ be the subsequence of $w_1, w_2, \ldots, w_s$ visited by procedure findPairs in the while loop on lines 4-9. Note that once tree $UT(bp(u_j, b))$ is marked as visited in line 8 it is never processed again in any subsequent calls of combineUnionSubtree on line 7. This is because next node u_{j+1} is ancestor of u_j and by property

3.1 either $UT(bp(u_{j+1}, b)) = UT(bp(u_j, b))$ or $UT(bp(u_j, b))$ is subtree of $UT(bp(u_{j+1}, b))$, and therefore marking of $UT(bp(u_{j+1}, b))$ also excludes $UT(bp(u_j, b))$.

(if) Let (p_1, p_2, l) be a maximal repeat in S such that $l \geq k$. Since (p_1, p_2, l) is right maximal, by lemma 2.1 $lcplen_T(w) = l \geq k$ and therefore $s \geq 1$ and $\exists j \in \{1..s\} : w = w_j$. We need to show that also $\exists j \in \{1..p\} : w = u_j$. If $w = v_1$ this holds for $j = 1$. Let's consider case $w \neq v_1$. Let i be maximal index such that w is ancestor of u_i. It holds that either $w = u_{i+1}$ or u_{i+1} is ancestor of w. $u_{i+1} = up(u_i, a_1)$ therefore $UP(u_{i+1}, u_i, a_1)$ and $DUP(u_{i+1}, u_i, a_1)$. We also have $(w, u_i) \in E^+(T)$ and $b^+(w, a_2) \supset b^+(u_i, a_2)$, because w is lowest ancestor of v_1 in whose union tree p_2 occurs. $a_1 \neq a_2$ because (p_1, p_2, l) is left maximal repeat. From this we have $UP(w, u_i, a_1)$. If u_{i+1} is ancestor of w then $depth_T(u_{i+1}) < depth_T(w)$, but that contradicts $DUP(u_{i+1}, u_i, a_1)$. Therefore $u_{i+1} = w$. When node w is visited by while loop on lines 4-9, combineUnionSubtree(w, $bp(w, a_2)$, $visited[a_2]$, l) is called. $bp(w, a_2) \neq visited[a_2]$ because $w = LCA_T(v_1, v_2)$. p_2 occurs in a bucket of $UT(bp(w, a_2))$ and therefore pair (p_2, l) is reported.

(only if) Let (p_2, l) be a pair reported by a call findPairs(p_1,k). It had to be reported by a call of procedure combineUnionSubtree on u_j for some j in line 7 of findPairs. From condition of while loop on line 4 we have $l \geq k$. The pair (p_2, l) could only be reported if $a_2 \neq a_1$, because of condition in line 6. u_j is common ancestor of v_1 and v_2. We will show that it is also lowest common ancestor w. We will prove $u_j = w$ by contradiction. Let $u_j \neq w$. Then u_j is ancestor of w because w is first to contain p_2 in it's union tree and w is ancestor of u_{j-1} because $UT(bp(u_{j-1}, a_2))$ can't contain p_2. If it did, p_2 would be reported in call combineUnionSubtree on u_{j-1}, subtree $UT(bp(u_{j-1}, a_2))$ would be marked as visited and the call combineUnionSubtree on u_j wouldn't be able to report p_2. as $u_j = up(u_{j-1}, a_1)$, we have $UP(u_j, u_{j-1}, a_1)$ and $DUP(u_j, u_{j-1}, a_1)$. From properties of w above, also $UP(w, u_{j-1}, a_1)$ and $depth_T(w) > depth_T(u_j)$ which contradicts $DUP(u_j, u_{j-1}, a_1)$.

Since $u_j = w$ it holds that $l = lcplen_T(w)$ and since $w = LCA_T(v_1, v_2)$, by lemma 2.1 we have (p_1, p_2, l) is right maximal repeat. Therefore (p_1, p_2, l) is maximal repeat in S with $l \geq k$. □

Theorem 3.3. *findPairs runs in time $O(z)$ where z is number of reported pairs.*

Proof. Let $a_1 = LC_S(p_1)$. Let $u_1 = map_T(p_1)$, $u_2 = up(u_1, a_1)$, $\ldots$, $u_p = up(u_{p-1}, a_1)$ be all nodes visited in while loop on lines 4-9.

Let's consider set C of all calls to combineUnionSubtree from line 7 in findPairs. Let C_a be subset of calls that report at least one pair and C_b subset of calls that don't report any pair because they are called on visited node. By lemma 3.7, total time t_a spent in all C_a calls is $O(z)$. Total time t_b spent by C_b calls is $O((p-1)|\Sigma|)$. (if a call occurs in u_1, it is a C_a call.)

From node u_i, we continue to node $u_{i+1} = up(u_i, a_1)$ if it exists. We know that, $\exists b \in \Sigma_{\mathfrak{e}}, b \neq a_1 : b^+(u_{i+1}, b) \supset b^+(u_i, b)$. This means that $bp(u_{i+1}, b)$ is ancestor of $bp(u_i, b)$ in $UT(bp(u_{i+1}, b))$ and can't have been visited before and $UT(bp(u_{i+1}, b))$ has therefore at least one new bucket that will be visited by combineUnionSubtree.

Thus for each node in $\{u_2, \ldots, u_p\}$ at least one C_a call is made and $(p-1) \leq |C_a| \leq z$. t_b is therefore $O(z)$ too. Initialisation in lines 1-3 takes constant time and time spent in each node for other purpose than for C calls is also constant. Total time taken by findPairs is therefore $O(z)$. □

3.6 Properties of R3 tree

Lemma 3.8. *Let $T = R3T(S)$, $T_1 = LCPIT(S)$, $T_1^R = LCPIT(S^R)$, $\boldsymbol{I} \in V(T_1) = V(T)$, $\boldsymbol{R} = rev(\boldsymbol{I})$. Then*

- *If $\boldsymbol{R}.lcp > \boldsymbol{I}.lcp$ then $bpsize_T(\boldsymbol{I}) = 1$*
- *If $\boldsymbol{R}.lcp = \boldsymbol{I}.lcp$ then $bpsize_T(\boldsymbol{I}) \leq |Children_{ST(S^R)}(\boldsymbol{R})|$*

Proof. Let $\boldsymbol{R}.lcp > \boldsymbol{I}.lcp$. This means that $\exists a : \forall p \in Pos^+_{T_1^R}(\boldsymbol{R}) : S^R[p + \boldsymbol{I}.lcp] = a$, which means that all occurences of $prefix_{T_1}(\boldsymbol{I})$ in S have left context a, which means $bpsize_T(\boldsymbol{I}) = 1$. Let $\boldsymbol{R}.lcp = \boldsymbol{I}.lcp$. Then $prefix_{T_1}(\boldsymbol{I})^R = prefix_{T_1^R}(\boldsymbol{R})$.

$$\forall p \in Pos^+_T(\boldsymbol{I}) : \big(LC_S(p) = a \Rightarrow S^R[revpos_S(p, \boldsymbol{I}.lcp)..revpos_S(p, \boldsymbol{I}.lcp) + \boldsymbol{I}.lcp] = a\big)$$

This means that if $bp(\boldsymbol{I}, a) \neq \emptyset$ then there is a-edge outgoing from $\boldsymbol{R}$ in $ST(S^R)$ which means $bpsize_T(\boldsymbol{I}) \leq |Children_{ST(S^R)}(\boldsymbol{R})|$. □

Theorem 3.4 shows that volume of information stored in bp-tables is not dependent on $|\Sigma|$.

Theorem 3.4. *Let $T = R3T(S)$, $n = |S| + 1$. Then*

$$\sum_{v \in V(T)} bpsize_T(v) < 3n.$$

Proof. Let $T_1 = LCPIT(S)$, $T_1^R = LCPIT(S^R)$. Let's partition $V(T)$ into $V_1 = \{v \in V(T) | rev(v).lcp = v.lcp\}$ and $V_2 = \{v \in V(T) | rev(v).lcp > v.lcp\}$. Let $rev(V_1) = \{rev(v) | v \in V_1\} \subseteq V_I(ST(S^R))$.

$$(1)$$

$$\sum_{v \in V_1} bpsize_T(v) \overset{\text{lemma 3.8}}{\leq} \sum_{v \in V_1} |Children_{ST(S^R)}(rev(v))| \overset{\text{lemma 2.2}}{=} \sum_{v \in rev(V_1)} |Children_{ST(S^R)}(v)|$$
$$\leq \sum_{v \in V_I(ST(S^R))} |Children_{ST(S^R)}(v)| \overset{\text{prop. 2.2}}{<} 2n$$

$$(2)$$

$$\sum_{v \in V_2} bpsize_T(v) \overset{\text{lemma 3.8}}{=} |V_2| \overset{\text{prop. 2.1}}{<} n$$

Now we have

$$\sum_{v \in V(T)} bpsize_T(v) = \sum_{v \in V_1} bpsize_T(v) + \sum_{v \in V_2} bpsize_T(v) \overset{(1),(2)}{<} 3n.$$

□

Theorem 3.5 shows that volume of information stored in up-tables is not dependent on $|\Sigma|$. Each up-table contains at most two distinct values.

Theorem 3.5. *Let* $T = R3T(S)$, $v \in V(T)$, $a, b, c \in \Sigma_{\mathfrak{c}}$, $a \neq b$, $b \neq c$, $c \neq a$, $u_a = up(v, a)$, $u_b = up(v, b)$, $u_c = up(v, c)$. *Then*

$$u_a = u_b \vee u_b = u_c \vee u_c = u_a$$

Proof. Let's assume $u_a \neq u_b \wedge u_b \neq u_c \wedge u_c \neq u_a$. All of nodes u_a, u_b and u_c lie on the path from v to $root(T)$. W.L.O.G, let's suppose that

$$depth_T(u_a) < depth_T(u_b) < depth_T(u_c) < depth_T(v)$$

From definition of up function we have

- $UP(u_a, v, a) \Rightarrow \exists d_a \in \Sigma_{\mathfrak{c}} : d_a \neq a \wedge b^+(u_a, d_a) \supset b^+(v, d_a)$
- $UP(u_b, v, b) \Rightarrow \exists d_b \in \Sigma_{\mathfrak{c}} : d_b \neq b \wedge b^+(u_b, d_b) \supset b^+(v, d_b)$
- $UP(u_c, v, c) \Rightarrow \exists d_c \in \Sigma_{\mathfrak{c}} : d_c \neq c \wedge b^+(u_c, d_c) \supset b^+(v, d_c)$

Now we consider two cases

- $d_c = a$ - this violates condition $DUP(u_b, v, b)$.
- $d_c \neq a$ - this violates condition $DUP(u_a, v, a)$.

□

Chapter 4

R3 tree implementation

This chapter will show how R3 tree can be implemented and describes algorithms and data structures that are used in R3Lib library. First we will show how R3 tree is stored in computer memory. We will describe it's data structures and their properties and present version of findPairs that works over these data tructures. Then we will show how R3 tree in this representation can be constructed. R3Lib works with arbitrary binary data. Our alphabet will be alphabet of bytes, $\Sigma = \{0, \ldots, 255\}$. Some details about representation of special symbol ¢ are still hiden to make algorithms more legible. We suppose we work with strings with length $n < 2^{30} - 1$. This allows us to represent every number in $\{0, \ldots, 2^{30} - 2\}$ with 32 bit integer where two bits are left for other purposes. We will often use these flags in our representation. It also leaves us one value $2^{30} - 1$ for special purposes, which will be denoted $\perp$.

4.1 Representation of Union forest

Union tree will be represented in single array of integers. Item i will have associated value[i] and flag[i] indicating whether it is *value item* (flag[i] = 0) or *navigator item* (flag[i] = 1).

There are two types of navigator items: *B-navigator* items and *U-navigator* items. B-navigator item represents a bucket from $B(T)$. For a B-navigator item k it holds that item at $k-1$ is value item. U-navigator item represents an union node from $B^u(T)$. For U-navigator item k it holds that item at $k-1$ is a navigator item.

Each bucket from $B(T)$ is stored as continuous interval of value items followed by one B-navigator item. value[k] of navigator item at index k is index of first value item of it's bucket. To retrieve a bucket from this array, one needs to know index k of it's navigator item. All positions of this bucket are then value[value[k]], value[value[k]+1], ..., value[$k-1$].

Sets from $B^u(T)$ will be represented by continuous sequence of subtrees, from which it is composed and one U-navigator item. value[k] of U-navigator item k is index of first value item, that belongs to it's subtree. Interval corresponding to U-navigator item spans over multiple subtrees and the items on indices value[k], value[k]+1, ..., $k-1$ may be also navigator items.

This representation requires nodes of union tree to be added in bottom-up manner. If an union tree node B_i is represented by navigator item i, node B_j is represented by navigator

item j and B_i is descendant of B_j, then $i < j$. Algorithm that builds union forest this way will be presented later.

By lemma 3.5, number of U-navigator items in each subtree is smaller than number of B-navigator items. This means that we can retrieve all positions from union node represented by U-navigator item k by scanning the array from index value[k] to $k-1$, reporting value items and skipping navigator items. This operation will cost us $O(|B|)$ time, for set $B \in B^+(T)$.

Union forest is represented by an array composed from arrays for it's union trees appended together. In union forest for a suffix tree $T = ST(S)$, $|S| + 1 = n$, there are exactly n value items. There is exactly one B-navigator item for each bucket and one U-navigator item for each union node. Number of B-navigator items is smaller than number of value items and number of U-navigator items is smaller than number of B-navigator items. Therefore total worst-case size of the array is $3n$, which means $12n$ bytes. Example of union forest in this representation is shown on figure 4.1 (array UF).

4.2 Representation of R3 tree

As we could see in previous chapter, R3 tree is basically internal part of a suffix tree (which is isomorphic with lcp-interval tree) with *bp* pointers to union forest of that tree and *up* pointers for quicker navigation of findPairs algorithm.

R3 tree will be represented by a table of nodes (lcp-intervals) and some additional arrays. Table of nodes will have following fields: LCP, LC1, UP1, UP2, BP.

Let $T = R3T(S)$, $n = |S| + 1$. Entry at index i in node table represents a node $v_i \in V(T)$.

- LCP[i] = $lcplen_T(v_i)$
- LC1[i] = $b \in \Sigma_{¢}$. We pretend that this value can be stored in one byte. (In R3lib, if $b = ¢$ we use auxiliary flag in UP1[i] value to indicate this.)
- UP1[i] = j, such that $v_j = up(v_i, a)$, $a \neq b$ (it also holds that v_j is parent of v_i, but we don't exploit this property in any way.)
- UP2[i] = j such that $v_j = up(v_i, b)$
- BP[i] = pointer to array BPTABLES. BP[i] is index of first item in bp-table for node v_i.

Additional arrays are BPTABLES, UF and MAP.

- BPTABLES - array that stores bp-tables for nodes. Each entry between index BP[i] and nearest index with end-flag represents pointer to an union forest node representing set $b^+(v_i, a)$ for some a. Left context a has to be found out by retrieving a position from the set and finding it's left context. Each entry has one additional flag 'own' that indicates that $b(v_i, a) \neq \emptyset$. Let BTABLES[j] = k and own[j] = 1. If UF[k] is an B-navigator item, then $b(v_i, a)$ is represented by this item. If UF[k] is an U-navigator node, then $b(v_i, a)$ is represented by item $k - 1$.

- UF - contains union forest as described in previous section.
- MAP[i] contains value j such that $v_j = map_T(i)$.

Worst case size of this representation is $45n$ bytes ($+\ n$ bytes of original string). This amount is distributed among particular arrays the following way

- node table - $17n$ ($4 \times$ integer $+\ 1 \times$ byte per entry, $|V(T)|$ entries, $|V(T)| < n$)
- BPTABLES - $12n$ ($3n$ integers)
- UF - $12n$ ($3n$ integers)
- MAP - $4n$ (n integers)

Snapshot of these memory structures, built for simple input is shown on figure 4.1. Bp-tables in array BPTABLES are divided by horizontal lines and end-flag is hidden. Left context of union trees is present for clarity, but it isn't really stored in array UF. Navigator items in array UF are highlighted. U-navigator items are darker than B-navigator items.

FindPairs query working with this representation is shown on figure 4.2. To determine left context of position p, in pseudo-code denoted LC(p), we have to have access to input string.

There is small technical difference between representation described here and implementation of R3lib regarding representation of union tree with left context ¢ (bucket with position 0). This union tree is not explicitly represented in array UF. We use auxilliary flags in value LCP[i] to indicate situations $b(v_i, ¢) = \{0\}$ and $b^+(v_i, ¢) = \{0\}$.

4.3 Construction of R3 tree

In this section we will show how to build R3 tree in representation described above. Following text should be read as extended comment for source code of R3lib. All R3 tree related functions are implemented in file `r3.c`. When we want to create R3 tree, we call function `r3t_create`. The construction process can be divided into several phases. For each phase, `r3t_create` calls different function and each of them can be completed in time linear in input data size. We explain each construction phase in details. Time complexity will be discussed only for selected phases or subroutines, where time requirements are not obvious.

Most of information contained in R3 tree comes from suffix tree. To build bp-tables and union forest we need to have it's structural information available. To avoid the complex operation of building suffix tree, we've chosen approach of [2], where it is shown that for simulating bottom-up traversal of suffix tree $ST(S)$ we need only suffix array and lcp-table (two integer arrays of length $|S| + 1$). This will be enough to build arrays LCP, BP, BPTABLES and UF. During this phase we also extract parent-child information from the suffix tree into temporary array FD. This array will be used to compute UP1 array required for top-down traversal and computation of arryas UP2 and LC1. In the last phase we compute MAP array. This is summary of R3 tree construction phases:

1) suffix array and lcp-table construction

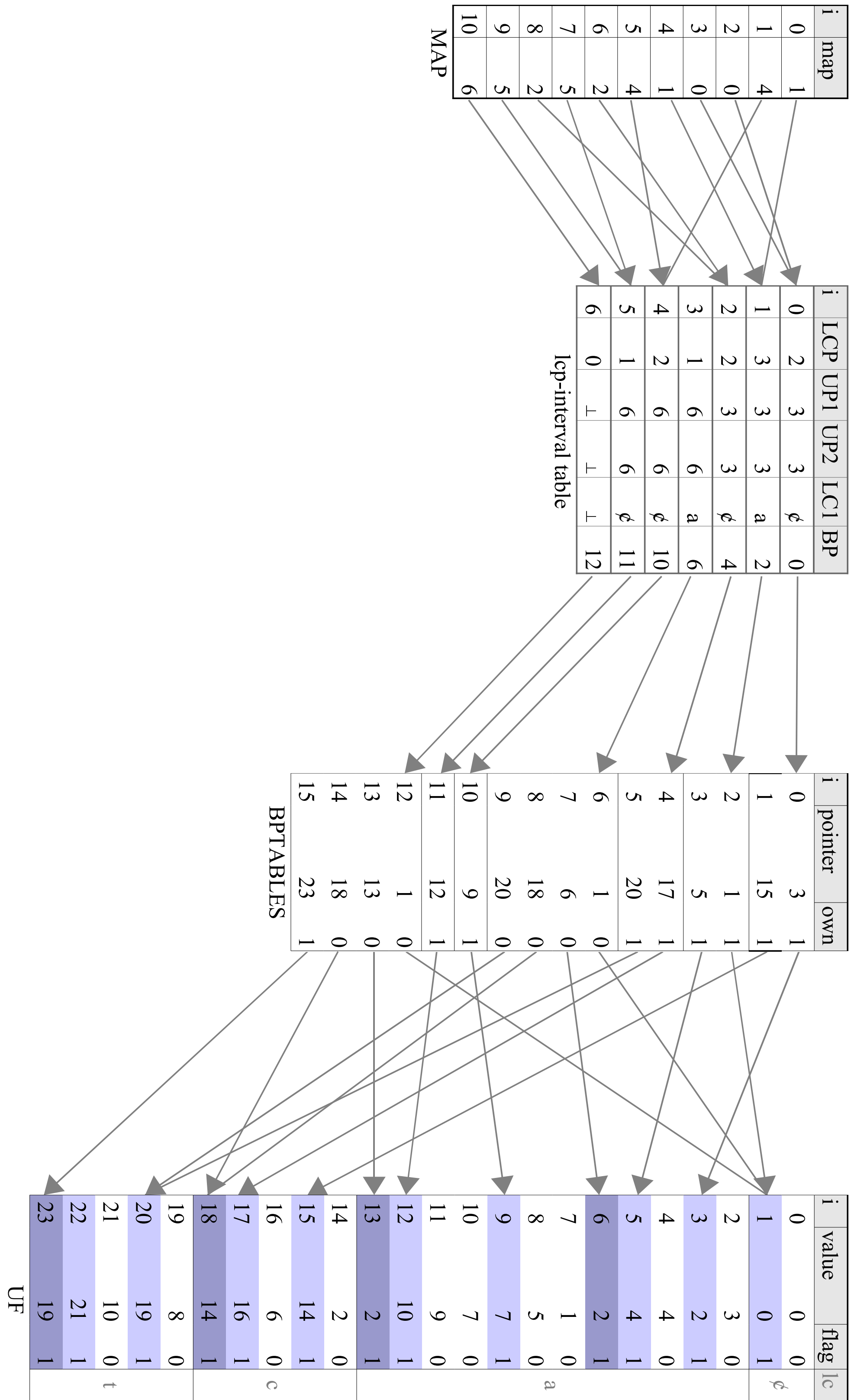

Figure 4.1: Data structures of R3 tree for string 'acaaacatat$.

```
combineUnionSubtree(k,visited,l)
1     if k = visited exit
2     for i := k-1 downto UF[k].value
3           if UF[i].flag = 0
4                 add pair (l, UF[i].value)
5           else
6                 if i = visited
7                       i = UF[i].value -1
```

```
findPairs(p, k)
1     for each a ∈ Σ_c
2           visited[a] := ⊥
3     v := MAP[p]
4     while v ≠⊥ ∧ LCP[v] ≥ k
6           i := BP[v]-1
7           repeat
8                 i := i+1
9                 j := BPTABLES[i].pointer
10                fd := UF[j].value
11                lc := LC[UF[fd].value]
12                if lc ≠ LC(p)
13                      combineUnionSubtree(j, visited[lc], LCP[v])
14                      visited[lc] := j
15          until BPTABLES[i].endflag = 1
16          if LC1[v] ≠ LC(p)
17                v := UP1[v]
18          else
19                v := UP2[v]
```

Figure 4.2: Algorithm 3 - findPairs query with new representation.

2) bottom-up traversal (LCP, BP, BPTABLES, UF, FD)

3) UP1 construction

4) top-down traversal (LC1, UP2)

5) MAP construction

4.3.1 Phase 1: suffix array and lcp-table construction

Suffix array and Lcp-table can be constructed in linear time and space. In R3lib, suffix array is computed using algorithm of [6]. We've modified the original code of Pang Ko, so that it would compute suffix arrays over arbitrary binary data. Lcp-table is build independently afterwards using algorithm of [9] with space saving tricks described in [10]. The memory requirements of the whole process do not exceed $10n$. See functions `createSuffixArray` and `createLCPTable` in file `suffixArray.c`.

4.3.2 Phase 2: bottom-up traversal (LCP, BP, BPTABLES, UF, FD)

Authors of [2] present algorithm (Algorithm 4.4, p.63) that simulates bottom-up traversal of lcp-interval tree (internal part of suffix tree). It calls function *process* on each encountered lcp-interval. By specifying function process we can implement construction steps needed in the second phase. For each lcp-interval we will need to

- create new entry in lcp-interval table
- add lc-buckets to union forest
- create bp-table for lcp-interval

For our purposes, this algorithm needs a small modification, which is shown as Algorithm 4 on fig. 4.3. Original version works with tuples $\langle lcp, rb, lb, childList\rangle$, where lcp is lcp-value of the lcp-interval, lb resp. rb is left resp. right boundary of lcp-interval and $childList$ is list of children intervals of the lcp-interval. We will use stack of 4-tuples $\langle lcp, lb, rb, fd\rangle$. Each tuple represents an lcp-interval with additional field fd - first descendant. This information will be used to determine parent-child relationships of lcp-intervals in our table. Operation push inserts one item on the top of the stack. Operation pop removes one item from top of the stack. top is pointer to the top of the stack. We acces components of tuple $\mathrm{I} = \langle lcp, lb, rb, fd\rangle$ by I.lcp, I.lb, I.rb, I.fd .

Following lemma will be presented without proof, for details see [2].

Lemma 4.1. *Algorithm 4 calls process() on tuple* $\langle lcp, lb, rb, fd\rangle$ *if and only if* (lcp, lb, rb) *is an lcp-interval. Intervals are processed in bottom-up fashion i.e. if lcp-interval* (lcp_1, lb_1, rb_1) *is descendant of* (lcp_2, lb_2, rb_2) *then tuple* $\langle lcp_1, lb_1, rb_1, fd_1\rangle$ *is processed before* $\langle lcp_2, lb_2, rb_2, fd_2\rangle$.

This algorithm is implemented in function `r3th_traverseBottomUp` and it runs in linear time (see [2] for details) if we assume that function *process* runs in constant time.

```
1   cnt := 0
2   lastInterval := ⊥
3   push ⟨0, 0, ⊥, 0⟩
4   for i = 1 to n-1 do
5       lb := i-1
6       while lcp[i] < top.lcp
7           top.rb := i-1
8           lastInterval := pop()
9           process(lastInterval)
10          lb := lastInterval.lb
11          cnt := cnt + 1
12          if lcp[i] = top.lcp
13              lastInterval := ⊥
14      if lcp[i] > top.lcp
15          if lastInterval ≠⊥ then
16              push ⟨lcp[i], lb, ⊥, lastInterval.fd⟩
17          else push ⟨lcp[i], lb, ⊥, cnt⟩
18  while stack not empty
19      top.rb = n-1
20      process(pop())
```

Figure 4.3: Algorithm 4 - computing FD and processing lcp-intervals

Now, let's take a look at function *process*. This function is too technical to be presented in pseudo code. This text will only serve as extended comment for the source code of function `r3th_processIntervalBottomUp`.

Let T be R3 tree to be constructed. Let $R_0, R_1, \dots, R_{m-1}$ be the sequence of all tuples in order they were processed on line 9 by Algorithm 4. When tuple R_i is being processed, lcp-interval table of T already contains valid values in fields LCP, FD and BP up to index $i-1$. For LCP and FD it holds that $\forall j, 0 \leq j < i : LCP[j] = R_j.lcp \land FD[j] = R_j.fd$. Let's say that $\forall k, 0 \leq k < m$, tuple R_k represents lcp-interval (or R3 tree node) $v_k \in V(T)$. From now on, we'll refer to lcp-interval represented by row i in node table by v_i.

Array FD will be used to determine parent-child relationship of lcp-intervals. This will be done by exploiting following property:

$$\forall j : v_j \text{ is descendant of } v_i \text{ if } FD[i] \leq j < i.$$

We know that all descendants of v_i have been processed before v_i and every node that belongs to subtree of v_i (if any) is on some index between $FD[i]$ (first descendant of v_i) and $i-1$. To determine only children of v_i instead of all descendants, we have to skip child subtrees as shown on fig. 4.4.

```
getChildren(i)
1    j := i-1
2    while j ≥ FD[i]
3         report child j
4         j := FD[j]-1
```

Figure 4.4: Retrieving children of lcp-interval v_i

We can also access sets $b(u,a)$ and $b^+(u,a)$ for each child u of v_i and for each $a \in \Sigma_{\mathfrak{c}}$, because arrays BP, BPTABLES and UF already contain correct data for children of v_i. Main task of function process is to create (for each a) new union tree $UT(b^+(v_i,a))$ by composition of union trees of children of v_i and possibly adding bucket $b(v_i,a)$ (if it's not empty).

For this purpose we compute auxilliary array *firstTree*. Let's consider following two cases:

1) For all children u of v_i, $b^+(u,a) = \emptyset$
   In this case firstTree[a] $= \perp$. Union tree $UT(b^+(v_i,a))$ consist only of bucket $b(v_i,a)$ if this bucket is non-empty. In such case index of it's B-navigator item is stored in BPTABLES.

2) There are children $v_{j_1}, v_{j_2}, \dots, v_{j_c}$ of v_i such that $j_1 < j_2 < \dots < j_c \land \forall k, 1 \leq k \leq c : b^+(v_{j_k},a) \neq \emptyset$.
   Let $v_{j_1}, v_{j_2}, \dots, v_{j_c}$ be all such children. firstTree[a] is then index of navigator item for $b^+(v_{j_1},a)$ in UF. If $c > 1$ or $b(v_i,a) \neq \emptyset$ we will have to create new U-navigator item in UF to union all subtrees into new union tree. This item will be added after all previous subtrees and possibly one $b(v_i,a)$ bucket. If k is index of new U-navigator item, then

we put $UF[k].value := UF[firstTree[a]].value$. The situation is illustrated on fig. 4.5. New index k is then stored in array BPTABLES.

In both cases, if $b(v_i, a) \neq \emptyset$, we set 'own' flag to 1 in respective BPTABLES entry. Buckets $b(v_i, a)$ are computed by bucketing elements of $Pos_T(v_i)$ by their left context. The set $Pos_T(v_i)$ is determined by skipping Pos^+ sets of children lcp-intervals and processing only suffix array items that, weren't processed before. This takes $O(|\Sigma|)$ time, because $Pos_T(v_i) \leq |\Sigma|$ and $|Children_T(v_i)| \leq |\Sigma|$. Computing *firstTree* array takes $O(|\Sigma^2|)$ time. Function `r3th_processIntervalBottomUp` therefore runs in time $O(|\Sigma^2|)$ and this means constant time for us.

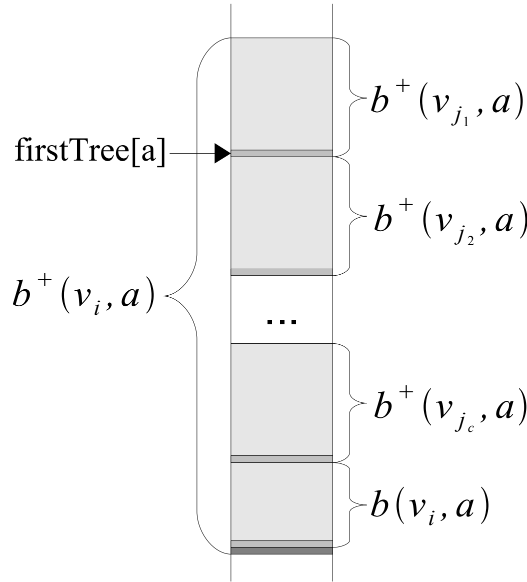

Figure 4.5: Composition of union trees of children of v_i into union tree of v_i

The array BPTABLES is built by adding one bp-table at the current end of array in each call to function *process*. The array UF is composed from segments of particular union trees. These segments are built in parallel, because one call to *process* may need to add buckets to union trees with different left contexts. Before start of phase 2, array UF have been allocated for $3n$ items (in function `r3th_allocate`), space for each segment is allocated according to count of symbols in input string S and pointers to each union tree are set (in table `r3tw->ut_ptr`). When phase 2 is completed, array UF may contain gaps between union tree segments. That's why phase 2 is followed by auxilliary procedure `r3th_compactUnionForest`, that removes unused space and compacts array UF.

4.3.3 Phase 3: UP1 construction

Algorithm that transforms FD array into UP1 array is shown in fig. 4.6. It uses stack of integers with classical top, push and pop operations. Array UP1 is constructed so that UP1[i] stores index of parent of v_i. This phase is implemented in function `r3th_computeParent`.

4.3.4 Phase 4: top-down traversal (LC1, UP2)

Let m be number of nodes in node table, created by bottom-up traversal. For $0 \leq i < m$, let v_i be node represented by i-th table row. Pseudo code for top-down traversal at fig. 4.7 uses function findBpEntries that is defined only in theoretical terms for simplicity. Function findBpEntries can be computed in constant time, because it works with constant sized bp-tables and property $b^+(v_i, a) \supset b^+(v_j, a)$ can be tested in constant time with our representation of union trees. The whole phase can be therefore finished in $O(m)$ time. Implementation can be found in function `r3th_traverseTopDown`.

4.3.5 Phase 5: MAP construction

Algorithm on fig. 4.8 visits every node v_i in node table of R3 tree T and goes through each entry in bp-table for node v_i. For each position $p \in b(v_i, a)$ sets MAP[p] := i. Algorithm makes $m = V(T)$ node visits. Since $B(T)$ is partition of the set $N_{|S|+1}$, assignment on line 10 is made for each position exactly once so there are exactly $|S| + 1$ executions of this assignment. This means that phase 5 can be completed in time $O(m + n)$ or $O(n)$. Implementation can be found in function `r3th_computeMap`.

```
computeParent()
1    UP1[m-1] := ⊥
2    push(m-1)
3    for i := m-2 downto 0 do
4        UP1[i] := top
5        push(i)
6        while stack not empty and FD[top] = i
7            pop()
```

Figure 4.6: UP1 construction

```
findBpEntries(i, j)
1     let a be such that b+(v_i, a) ⊃ b+(v_j, a) (such a has to exist)
2     u_1 = index of navigator item for b+(v_i, a)
3     if (∃b)(b ≠ a) : b+(v_i, b) ⊃ b+(v_j, b)
4           u_2 = index of navigator item for b+(v_i, b)
5     else
6           u_2 =⊥
7     return (u_1, u_2)

traverseTopDown
1     UP1[m-1]=⊥
2     UP2[m-1]=⊥
3     LC1[m-1]=¢
4     for i := m-2 downto 0 do
5           parent := UP1[i]
6           (u_1, u_2) := findBpEntries(parent,i)
7           u := UF[u_1].value
8           LC1[i] := LC(UF[u].value)
9           if u_2 =⊥
10                if(LC1[i] ≠ LC1[parent])
11                      UP2[i] := UP1[parent]
12                else
13                      UP2[i] := UP2[parent]
```

Figure 4.7: Top-down traversal of R3 tree

```
1 for i:= 0 to m-1
2     j := BP[i]-1
3     repeat
4           j := j+1
5           if BPTABLES[j].own = 1
6                 u := BPTABLES[j].pointer
7                 if UF[u-1].flag (u-navigator item at index u)
8                       u:=u-1
9                 for k := UF[u].value to u-1
10                      MAP[UF[k].value] := i
11    until BPTABLES[j].endflag = 1
```

Figure 4.8: MAP construction

Appendix A

R3lib documentation

R3lib 1.0.0 source code package contains following files

`test/analysis.c`	- tools for memory utilisation analysis and statistics
`test/analysis.h`	
`test/debug.c`	- debugging output tools
`test/debug.h`	
`test/rmalloc.c`	- external malloc debug library[1]
`test/rmalloc.h`	
`test/test.c`	- testing routines
`test/test.h`	
`test/test_r3.c`	- R3 tree specific tests
`test/test_r3.h`	
`test/test_sa.c`	- suffix array and lcp-table specific tests
`test/test_sa.h`	
`testdata`	- directory with basic test data
`array.c`	- simple augmentable array implementation
`array.h`	
`bitstr.c`	- bit array implementation
`bitstr.h`	
`conf.h`	- global configuration and definitions
`LICENSE.txt`	- GNU LGPL license file
`main.c`	- used only for running tests
`r3.c`	- R3 tree routines
`r3.h`	
`suffixArray.c`	- suffix array and lcp-table routines
`suffixArray.h`	

Files in directory `test` and `testdata` are used only during development of R3lib. They are not needed for regular use. R3 tree data structure is represented by structure `t_r3t` defined in file

[1] see http://www.hexco.de/rmdebug

r3.h. Main functionality of R3lib is covered by three functions r3t_create, r3t_findPairs and r3t_destroy that are defined in file r3.c. By calling function r3t_create we initialise the t_r3t structure. This function creates R3 tree for input data. This step is therefore most time consuming in the whole process. See Appendix B for preformance results. When R3 tree construction is finished, we can perform findPairs queries with function r3t_findPairs. When we don't need R3 tree anymore, clean-up should be made with function r3t_destroy, that frees memory allocated by r3t_create. Each step of this process is demonstrated in example function on fig. A.1 and A.2

Now we desrcibe each of the three functions in detail.

```
int r3t_create(t_r3t* r3t, BYTE* text, int textLen)
```

Creates R3 tree for text of length textLen (n) and initializes r3t structure. BYTE is defined as unsigned char. Data at location text may contain arbitrary BYTE values ($0 \dots 255$). Worst case peak memory usage is $52n$ bytes and worst case R3 tree size is $44n$ bytes (not counting the n bytes of text). Returns 0 on success, non-null value on error. See conf.h for error constants. See Appendix B for performance results.

```
int r3t_findPairs(t_r3t* r3t, int p1, int k, int* count,
                  int count_limit, int* p2, int* l)
```

Finds all pairs (p_2, l) such that (p1 , p_2, l) is maximal repeat and $l \geq$ k. Number of found pairs is returned in count. Pairs are stored as (p2[0], l[0]), (p2[1], l[1]), ..., (p2[count-1], l[count-1]). Pairs will be sorted by length, in descending order. Number of reported pairs can be limited by count_limit, to make sure, that function r3t_findPairs won't try to write more results to arrays p2 and l than they are allocated for.

```
int r3t_destroy(t_r3t* r3t)
```

Free all memory locations allocated by r3t_create called on r3t.

```c
#include <stdio.h>
#include <string.h>
#include "r3.h"

int example()
{
   int result, i;
   t_r3t r3t; // r3 tree structure
   int* p2; // storage for p2 component of (p2, l) pair
   int* l; // storage for l component of (p2, l) pair
   int pair_cnt; // number of found pairs
   int k; // minimal repeat length
   int p1; // first component of maximal repeat
   int limit; // maximal number of returned repeats

   /* input text */
   /* PATTERN occurs at positions 4, 16, 28 and 40 */
   BYTE* text = "abcdPATTERNabceaPATTERNbcfabPATTERNcgabcPATTERNhabc";

   /* size of input */
   int textLen = strlen(text);

   /* create r3 tree for given input */
   result = r3t_create(&r3t, text, textLen);
   if(result)
      return result;

   /* allocate space for query results */
   p2 = (int*) malloc(textLen*sizeof(int));
   l = (int*) malloc(textLen*sizeof(int));
   if(!p2 || !l)
      return -1;

   /* define parameters for findPairs query */
   p1 = 4; /* first component of maximal repeat */
   k = 7;  /* minimal repeat length */
   limit = textLen; /* maximal number of returned repeats */

   /* find all pairs (p2, l) such that (p1, p2, l) is
    * maximal repeat in text and l >= k */
   result = r3t_findPairs(&r3t, p1, k, &pair_cnt, limit, p2, l);
   if(result)
      return result;

   /* now we should have
    *
    * pair_cnt == 3
    * (p2[0], l[0]) == (16, 7)
    * (p2[1], l[1]) == (28, 7)
    * (p2[2], l[2]) == (40, 7)
    */
```

Figure A.1: Example function using R3lib

```
  printf("*** Example of findPairs query ***\n");
  printf("Text: '%s'\n", text);
  printf("Position: %i\n", p1);
  printf("Min. length: %i\n", k);

  for(i=0; i<pair_cnt; i++) {
    /* (p1, p2[i], l[i]) now represents maximal repeat */
    printf("found repeat: (%i, %i, %i)\n", p1, p2[i], l[i]);
  }

  /* free resources held by r3 tree structure */
  r3t_destroy(&r3t);
  return 0;
}
```

Figure A.2: Example function continued

Appendix B

Performance results

In this section, we supply some measurements to illustrate time and memory requirements for R3 tree construction. We've chosen various types of files, both text and binary as input data[1]. Table B.1 shows memory measurements for each file.

- File = file used as input data
- Size = size of input data in bytes
- Peak = peak memory usage during construction of R3 tree in bytes
- Mem = memory usage of finished R3 tree in bytes
- rPeak = Peak/Size
- rMem = Mem/Size

Time measurements were done for two operating systems: Windows XP and Linux. We've measured time taken to construct R3 tree four times for each file on each OS. Table B.2 shows average results for both operation systems. Machine used for measurement was 3GHz Intel Pentium 4, 1GB RAM.

- File = file used as input data
- Size = size of input data in bytes
- TimeWin = average time to construct R3 tree in seconds (OS: Windows XP SP1, gcc 3.4.4 cygwin)
- TimeLinux = average time to construct R3 tree in seconds (OS: Linux Ubuntu 6.06, Kernel 2.6.15-27-386, gcc 4.0.3)

[1] At the time when the measurements took place all of these files were free to download. Links to these files can be found on R3lib project home page: http://michal.linhard.sk/r3lib

Table B.1: Memory measurements.

File	Size	Peak	Mem	rPeak	rMem
gimp-2.2.9.tar.gz	18,400,967	691,693,912	416,358,967	37.59	22.63
Azureus_2.5.0.4a_Win32.setup.exe	10,007,784	362,287,916	206,977,004	36.20	20.68
180907088.jpg	4,611,403	164,415,064	92,187,563	35.65	19.99
183899999.jpg	1,993,582	69,729,972	38,079,982	34.98	19.10
184722357.jpg	1,065,155	36,752,568	19,964,923	34.50	18.74
rbcru10.txt	653,492	27,613,492	16,417,866	42.26	25.12
zlib123.zip	583,873	20,627,024	11,447,525	35.33	19.61
2ws2610.txt	193,004	8,072,116	4,839,906	41.81	25.07
1ws2510.txt	150,662	6,355,132	3,797,228	42.18	25.20
1ws1810.txt	145,500	6,125,588	3,647,890	42.10	25.07
1ws3410.txt	129,092	5,435,572	3,245,222	42.11	25.14
1ws1710.txt	121,446	5,114,920	3,053,803	42.12	25.15
1ws0610.txt	105,421	4,445,260	2,651,660	42.17	25.15
19033.txt	74,726	3,178,480	1,916,273	42.54	25.64

Table B.2: Time measurements.

File	Size	TimeWin	TimeLinux
gimp-2.2.9.tar.gz	18,400,967	149.701	174.265
Azureus_2.5.0.4a_Win32.setup.exe	10,007,784	57.460	67.203
180907088.jpg	4,611,403	22.031	25.509
183899999.jpg	1,993,582	7.649	8.850
184722357.jpg	1,065,155	3.080	3.617
rbcru10.txt	653,492	3.364	4.073
zlib123.zip	583,873	1.756	2.116
2ws2610.txt	193,004	0.898	1.098
1ws2510.txt	150,662	0.712	0.892
1ws1810.txt	145,500	0.673	0.831
1ws3410.txt	129,092	0.597	0.734
1ws1710.txt	121,446	0.560	0.695
1ws0610.txt	105,421	0.480	0.585
19033.txt	74,726	0.353	0.396

Bibliography

[1] Brenda S. Baker: On finding duplication in strings and software, technical report, AT&T Bell Laboratories, February, 1993

[2] S. Kurtz, M.I. Abouelhoda and E. Ohlebusch: Replacing Suffix Trees with Enhanced Suffix Arrays, Journal of Discrete Algorithms, 2:53-86, 2004.

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

[4] P. Weiner: Linear pattern matching algorithms, in IEEE 14th Ann. Symp. on Switching and Automata Theory, 1973, pp. 1-11.

[5] E. Ukkonen: Constructing suffix trees on-line in linear time, in Algorithms, Software, Architecture. Information Processing 92, vol. I, Elsevier, 1992, pp. 484-492.

[6] P. Ko and S. Aluru: Space-efficient linear time construction of suffix arrays, Combinatorial Pattern Matching, pp. 200-210, 2003

[7] Juha Kärkkäinen and Peter Sanders: Simple linear work suffix array construction, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03). LNCS 2719, Springer, 2003, pp. 943-955

[8] D. K. Kim, J. S. Sim, H. Park, K. Park: Linear-time construction of suffix arrays, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, pp. 186-199, 2003

[9] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: Linear-time longest-common-prefix computation in suffix arrays and its applications. In Proc. 12th Symposium on Combinatorial Pattern Matching (CPM '01), pages 181-192. Springer-Verlag LNCS n. 2089, 2001.

[10] Giovanni Manzini: Two space saving tricks for linear time LCP computation, Technical Report TR-INF-2004-02-03-UNIPMN

[11] R. Giegerich and S. Kurtz: From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Algorithmica, 19:331-353, 1997.

[12] J. Stoye: Affix Trees. Technical Report 2000-04, Universität Bielefeld, Technische Fakultät, 2000.

List of Figures

2.1 Example of maximal repeat 5
2.2 Suffix tree for string S='acaaacatat' 7
2.3 Suffix array and lcp-table for S='acaaacatat'. 9
2.4 Lcp-interval tree for string S='acaaacatat' 10
2.5 Example of reverse lcp-interval in suffix array and lcp-table. 11

3.1 LC-bucket tree for string S='acaaacatat' 16
3.2 Algorithm 1 - Non-optimal findPairs query on LC-bucket tree 17
3.3 Visualisation of R3 tree for string S='acaaacatat' with union forest. 22
3.4 Another visualisation of R3 tree with union forest. 23
3.5 Algorithm 2 - Optimal findPairs query on R3 tree 24

4.1 Data structures of R3 tree for string 'acaaacatat$. 31
4.2 Algorithm 3 - findPairs query with new representation. 32
4.3 Algorithm 4 - computing FD and processing lcp-intervals 34
4.4 Retrieving children of lcp-interval v_i 35
4.5 Composition of union trees of children of v_i into union tree of v_i 36
4.6 UP1 construction 37
4.7 Top-down traversal of R3 tree 38
4.8 MAP construction 38

A.1 Example function using R3lib 41
A.2 Example function continued 42

Abstrakt

V tejto diplomovej práci predstavujeme dátovú štruktúru - R3 strom, ktorá efektívne reprezentuje maximálne opakovania v reťazci. R3 strom koncepčne vychádza zo suffixového stromu. Má lineárne pamäťové nároky a vieme ho skonštruovať v lineárnom čase aj priestore pre reťazce nad abecedou s konštantnou veľkosťou. Opakovanie v reťazci S definujeme ako trojicu (p_1, p_2, l), kde p_1, p_2 sú dve rôzne pozície v reťazci S a l je dĺžka opakovania. Dotazy na maximálne opakovania formulujeme vo forme funkcie $findPairs(p_1, k, S)$, ktorá vracia všetky dvojice (p_2, l) také, že (p_1, p_2, l) je maximálne opakovanie v reťazci S s dĺžkou $l \geq k$. R3 strom umožňuje výpočet dotazov findPairs v optimálnom čase $O(z)$, kde z je počet nájdených dvojíc.

Ďalšou dôležitou súčasťou práce je popis návrhu a funkcionality knižnice R3lib, napísanej v jazyku C. Táto knižnica slúži na vytváranie R3 stromov a vykonávanie findPairs dotazov nad reťazcami nad abecedou bajtov. To znamená, že umožňuje vyhľadávať maximálne opakovania v ľubovoľných textových ale aj binárnych súboroch.