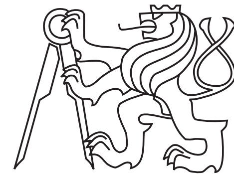
Sem vložte zadání Vaší práce.

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science



Master's thesis

Searching occurrences of tree patterns in ordered trees with the use of indexes of the trees

Bc. Jan Milík

Vedoucí práce: Doc. Jan Janoušek

13th May 2015

Acknowledgements

I would like to thank my friends with whom I lived past two years and who were an adoptive family to me. I would like to thank my family for being my actual family. Finally, I would like to thank my supervisor Doc. Janoušek for being patient with me.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act 60 no. 121/2000 (copyright law), and with the rights connected with the copyright act included the changes in the act.

In Prague 13th May 2015 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Jan Milík. All rights reserved.
This thesis is a school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Jan Milík. *Searching occurrences of tree patterns in ordered trees with the use of indexes of the trees: Master's thesis* . Czech Republic: Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstract

We compare two existing schemes for indexing trees, one based on a nondeterministic factor automaton, the other on deterministic compact suffix automaton. A third scheme is presented using position heaps – a relatively new data structures. As a side product, algorithm for converting suffix trees to position heaps and a new data structure based on the position heap is briefly sketched out. The three schemes are implemented and their running times measured. For most inputs, the third, position heap based scheme is found to be the fastest with minimal trade-off in the form of a small number of false positives.

Keywords pushdown, nondeterministic, bit-parallelism, automaton, machine, tree, subtree, pattern

Abstrakt

Tato práce porovnává dvě existující schémata pro indexování stromů. Jedno je založené na nedeterministickém faktorovém automatu, druhé na deterministickém kompaktním suffixovém automatu. Je zde popsáno třetí, nové schéma založené na pozičních haldách – relativně nové datové struktuře. Jako vedlejší produkt je popsán algoritmus pro převod suffixových stromů na poziční haldy a načrtnuta nová datová struktura založená na pozičních haldách. Všechna schéma byla implementována a jejich rychlost změřena. Pro většinu vstupů bylo třetí schéma založené na pozičních haldách shledáno nejrychlejším s minimálním cenou v podobě malého počtu falešných pozitiv.

Klíčová slova zásobník, nedeterminismus, bitový paralelismus, automat, strom, podstrom, vzorek

Contents

Introduction **1**

1 Basic notions and definitions **3**
1.1 Strings 3
1.2 Graphs and trees 3
1.3 Automata 6
1.4 Accepting trees and subtrees 8
1.5 String searching 9
1.6 Subtree patterns 17

2 WBC bitmap index **21**
2.1 Bit parallelism 21
2.2 Shift-And algorithm 21
2.3 Compact representation 22
2.4 Bit parallelism using run-length encoded bit vectors 23

3 Full and linear index **29**
3.1 Search Algorithm 30

4 Position heap index **35**

5 Implementation **39**
5.1 Basic architecture 39
5.2 Tools used 39
5.3 WBC index format 41
5.4 Bit vector representation 42
5.5 Factor automaton simulation 42
5.6 Full and linear index format 43
5.7 Position heap index format 45
5.8 Compact suffix automaton simulation 46
5.9 FLLI search function 48

6 On position heaps **49**

7 Experimental results **55**

Conclusion **59**
Analysis of the presented position heap conversion algorithms 59
New position heap-based data structure 59

Bibliography **61**

Index **63**

A Contents of the enclosed CD **65**

List of Figures

1.1 Two ordered trees differing only in ordering 4
1.2 Two distinct unranked ordered trees with the same prefix notation 5
1.3 Nondeterministic pattern matching finite state automaton 10
1.4 Factor automaton for the string "banana" 11
1.5 Conversions between suffix trees and automata 13
1.6 The suffix tree and the position heap for string "banana" 16
1.7 A ranked ordered tree 18
1.8 Nondeterministic tree pattern automaton for ranked prefix notation of the tree in fig. 1.7 18

2.1 Another ranked ordered tree 22
2.2 Layout of a WBC header 23
2.3 Layout of adapted WBC header 24
2.4 Bitwise AND of two WBC bit vectors 25
2.5 Run intersection finite automaton 26
2.6 Shift-And for two WBC bit vectors 26

3.1 Ranked ordered tree T_1 from example 1 31
3.2 Ranked prefix notation $\mathit{pref}(T_1)$ and compact suffix automaton $SA(\mathit{pref}(T_1))$ for tree T_1 from example 1 32
3.3 Subtree jump table SJT for FLLI index of tree T_1 from example 1 33

4.1 Position heap $PH(T_1)$ for tree T_1 from example 1 36

5.1 Abbreviated declaration of the `BitVector` class 44
5.2 Abbreviated declaration of the `WBCBitVector` class 45
5.3 Abbreviated declaration of the `Factor` structure 45
5.4 Abbreviated declaration of the `Factor` structure 46

6.1 Conversions between suffix index structures 50
6.2 Example: position heap for the string "abbac" and an automaton produced by the heap's minimization 52
6.3 Modified version of position heap-derived automaton with position labels placed at the edges 52

7.1 Subtree size frequency 56
7.2 Existing tree search, WBC algorithm only 56
7.3 Subtree pattern search using WBC, FLLI and position heap 57
7.4 False positives and actual occurrences in position heap index . . . 57

List of Tables

1.1 Trace of a nondeterministic forward pattern matching automaton . 11
1.2 Trace of a factor automaton 12

5.1 Primitive data types 42
5.2 `PString` data structure 42
5.3 `RankedSymbol` data structure 42
5.4 `WBCBitVector` data structure 43
5.5 Index file layout 43
5.6 Linear index file layout 47
5.7 `SAState` – Suffix automaton state layout 47
5.8 `SAEdge` – Suffix automaton edge layout 47
5.9 Suffix automaton layout 48

7.1 Datasets used for measurements 55

Introduction

The subject of this thesis is indexing trees and searching for occurrences of subtrees and subtree patterns. Subtree and tree pattern automata are one possible way of indexing trees and as such have wide range of applications such as XML databases, NOSQL databases, protein sequence databases and other scientific datasets, et cetera.

XML in particular has become ubiquitous today. Although it has been originally designed to markup documents and not necessarily structured data, it is commonly used for this purpose. It has also become a common format for the exchange of machine readable data over network the Internet and a common export format format databases of all kinds as is demonstrated, among other things, by the very datasets used in this work.

In fact, there is so much hierarchical data being produced every day in the world, it becomes useless without convenient and efficient way of indexing it all and searching it. Many modern solutions to this problem rely on relational databases or other indexing methods originally devised for flat data structures. This makes sense to a degree, since modern SQL databases are extremely optimized and well suited for handling very large data. However, by their nature, they must break queries over any tree structure down to many smaller relational queries, which makes them less efficient than is theoretically possible.

The goal of this work is to explore and compare a few tree indexing schemes that aim to provide better solution to the problem of indexing trees to allow for fast subtree and subtree pattern search.

More specifically, this thesis explores automata-based approaches, which take advantage of suitable ways of serializing trees into string to convert an arbological problem into a stringological one.

This work is the continuation and extension of [14].

Structure of this thesis

First, we'll discuss the basic notions, definitions and algorithms in Chapter 1. In Chapter 2, we will describe three schemes for indexing trees, each with a corresponding algorithm for searching for the occurrences of subtree patterns. We will then discuss some of the technical details of the implementation of

these algorithms in Chapter 5. We have measured the performance of all three algorithms and we will present the results along with some notes in Chapter 7. Finally, in the conclusion, we will mention some possible venues for future work on the subject of this thesis.

CHAPTER 1

Basic notions and definitions

1.1 Strings

Definition 1. Alphabet *is a finite non-empty set of symbols.*

Definition 2. Kleene closure *of alphabet* Σ *is the set of all finite strings over* Σ *and is denoted* Σ^**. Kleene closure always contains empty string,* ε*.*

Definition 3. *A* formal language L *over an alphabet* Σ *is some subset of* Σ^**. In mathematical notation,* $L \subseteq \Sigma^*$*.*

Consider an arbitrary alphabet Σ and a string $w \in \Sigma^*$.

Definition 4. *A string* f *over* Σ *is said to be a* factor of w *if* w *can be written as* $w = ufv$*, where* $u, v \in \Sigma^*$*.*

Definition 5. *A string* f *over* Σ *is said to be a* prefix of w *if* w *can be written as* $w = fv$*, where* $v \in \Sigma^*$*. The set of all prefixes of any* w *is subset of the set of all factors of* w*.*

Definition 6. *A string* f *over* Σ *is said to be a* suffix of w *if* w *can be written as* $w = uf$*, where* $u \in \Sigma^*$*. The set of all suffixes of any* w *is subset of the set of all factors of* w *and is denoted* $S(x)$*.*

1.2 Graphs and trees

The bulk of my work deals with *labeled, ranked, ordered, trees* written in preorder/prefix notation. *Ranked* means each node is "decorated" by a *rank*, also called *arity* and is represented by a *ranked symbol.* I define the arity of a node to be simply the number of the node's children. In graph theory

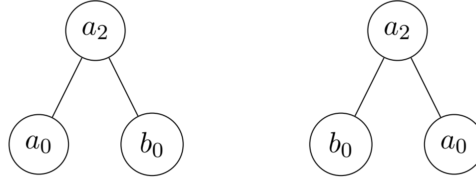


Figure 1.1: Two ordered trees differing only in ordering

language, the arity is the *outdegree* of the node. In this text, the arity of a node a is denoted $arity(a)$.

Ordered means the children of a node have defined order. For the sake of clarity, throughout this work the order of the nodes is always only implied by their spatial layout in the case of graphs, or by their symbols' order in the case of strings. The figure 1.1 shows two ordered trees that differ only in the order of some of its nodes. Despite the fact that the nodes have the same label and arity, the two trees are not considered equal.

Definition 7. *Ordered tree G is a tuple (N, E) where N is a set of nodes and E is a set $\{\forall f \in N \mid ((f, g_1 \in N), (f, g_2 \in N), \ldots, (f, g_n \in N));\ n = arity(e)\}$. Each 2-tuple represents an edge leaving f and entering $g_{1..n}$.*

Preorder notation is a way of expressing arbitrary trees as strings; linear data structures.

Definition 8. *The prefix notation of an ordered tree, $pref(f)$, is defined as:*

$$pref(f) = \begin{cases} pref(f) = f & \text{if } f \text{ is a leaf} \\ pref(f) = f\ pref(g_1)\ \ldots\ pref(g_n) & \text{where } g_{1..n} \text{ are the children of } f \end{cases}$$

The ranked prefix notation is defined in the same way and differs only in that it uses ranked symbols rather than plain node labels.

Having constructed some internal representation of a tree, the algorithm (see alg. 1) to construct prefix notation consists of traversing the tree in a depth-first preorder traversal and appending the current node's label to the result string.

Note that I describe the algorithm in terms of output stream rather than output string. This is to underline the fact that there is no need to keep any intermediary data in the memory apart from the input tree; the result can be written to a stream (such as opened index file) and immediately forgotten. If in-memory string is desired, however, a stream-like interface to a string can be easily provided in most environments.

For the prefix notation to be unambiguous (i.e. to be unique for a given tree), it is necessary to use ranked symbol. Using only plain labels, two distinct trees with equal number of nodes with the same labels may have the

Algorithm 1 Construction of preorder notation for a tree

input: A root node of a tree
output: Prefix notation of the tree written to a stream

```
init(outputStream)
init(stack)
push(stack, root)
while not empty(stack) do
    node ← pop(stack)
    write(outputStream, label(node))
    for each child in reversed(children(node)) do
        push(stack, child)
    end for
end while
```

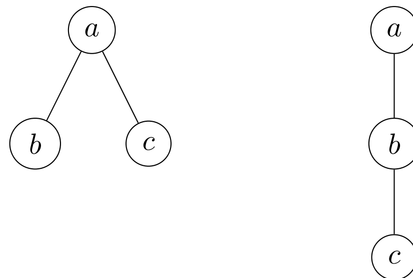


Figure 1.2: Two distinct unranked ordered trees with the same prefix notation

same prefix notation. See figure 1.2 for example. This also means that it is impossible to unambiguously reconstruct an unranked tree from simple prefix notation. To circumvent this problem, an alternate notation, the *bar notation* as defined in [10], can be employed and an algorithm (see 2) can be devised to convert the bar notation to ranked notation.

Definition 9. *The bar notation of an ordered tree, $bar(f)$, is defined as:*

$$bar(f) = \begin{cases} label(f)\ ] & \text{if } f \text{ is a leaf} \\ label(f)\ bar(g_1)\ \dots\ bar(g_n)\ ] & \text{where } g_{1..n} \text{ are the children of } f \end{cases}$$

Definition 10. *An alphabet $\mathcal{A}$ is a bar alphabet if it can be written as $\mathcal{A} = \mathcal{B} \cup \{]\}$, where $\mathcal{B}$ is some nonempty alphabet.*

It may be interesting to note that the Extensible Markup Language (XML) documents are basically trees written in bar notation. The XML closing tag, in the form `</tagName>`, is analogous to the bar symbol (]).

Algorithm 2 Conversion from bar prefix notation to ranked prefix notation

```
input: Input stream with the bar prefix notation
output: String with the ranked prefix notation
init(inputStream)
init(outputString)
init(parentStack)
index ← 0
while not at the end of inputStream do
    s ← readSymbol(inputStream)
    if s = ] then
        pop(parentStack)
    else
        parentIndex ← top(parentStack)
        arity(outputString[parentIndex]) ←
  arity(outputString[parentIndex]) +1
        push(parentStack, index)
        outputString ← concat(outputString, ranked(s, 0))
    end if
    index ← index +1
end while
```

1.3 Automata

Definition 11. Power set *of a set S is the set of all subsets of S including the empty set and S itself. Power set of S is denoted $P(S)$.*

Definition 12. Nondeterministic finite automaton *(NFA) is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, F)$ where*

- *Q is a finite set of states,*
- *Σ is a finite set of input symbols, also called the alphabet,*
- *Δ is a transition relation $\Delta : Q \times \Sigma \to P(Q)$,*
- *q_0 is the initial state $q_0 \in Q$,*
- *and F is a set of final states such that $F \subseteq Q$.*

A string $w = s_1\, s_2 \dots s_m$, $w \in \Sigma^*$ is said to be accepted by the M if there is a sequence $q_1\, q_2 \dots q_m \in Q^*$ such that $q_m \in F$ and for each q_i, the tuple (q_{i-1}, s_i, q_i) belongs to Δ.

Definition 13. Pushdown automaton *is a 7-tuple $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ where*

- *Q is finite set of states,*
- *Σ is a finite set of input symbols,*
- *Γ is a finite set of stack symbols,*
- *Δ is a transition relation $\Delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \to Q \times \Gamma^*$,*
- *q_0 is the initial state $q_0 \in Q$,*
- *Z is the initial stack symbol $Z \in \Gamma$,*
- *and F is a set of final states $F \subseteq Q$.*

Definition 14. Nondeterministic pushdown automaton *is a pushdown automaton $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ such that Δ is not a function.*

Definition 15. Deterministic pushdown automaton *is a pushdown automaton $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ such that Δ is a function.*

Definition 16. *Let $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ be a nondeterministic pushdown automaton. M is an* input-driven pushdown automaton *iff for each tuple $(s, \alpha) \in (\Sigma \cup \epsilon) \times \Gamma$ there is at most one $\beta \in \Gamma^*$ such that $\Delta(q, s, \alpha) = \{(r_1, \beta), (r_2, \beta), \ldots, (r_m, \beta)\}$ where $q \in Q$ and $r_1, r_2, \ldots, r_m \in Q$.*

This rather cryptic definition states simply that in each step of the simulation of an input-driven pushdown automaton the pushdown operation $\alpha \mapsto \beta$ is selected only by the current input symbol. Other way to phrase that is that the pushdown operations do not depend on the current state $q \in Q$.

The input-driven pushdown automata have the interesting property of always being determinisable. The determinisation is done in much the same way as for finite state automata. The algorithm is well-known and can be found in many sources including [13].

Definition 17. Counter automaton *is a pushdown automaton $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, F)$ such that $|\Gamma| = 1$.*

Counter automata are so called because to simulate the stack, a single integer counter is sufficient. This also implies that unlike the general pushdown automaton, counter automaton can be implemented with constant memory complexity in $|w|$, where w is the accepted string.

Definition 18. *Σ is an alphabet, M is any automaton accepting only the strings and all of the strings from formal language $L \subseteq \Sigma^*$. Then L is said to be the* language of *M and can be denoted $L(M)$.*

1.4 Accepting trees and subtrees

Ranked prefix notation allow any ordered tree T to be expressed as a string $w; w = pref(T)$ in such a way that the ranked prefix notation of any subtree S is a factor of w. Proof of this property can be found in [8] as proof to theorem 1. Bar notation has the same property. This allows wealth of string searching algorithms to be used to search for trees for subtrees in prefix notation.

Not all strings over a ranked alphabet or a bar alphabet need to be prefix notations of a tree, however. Consider, for instance, a ranked alphabet $\Sigma = \{a_0, a_3\}$ and a string over this alphabet $p = a_3\ a_0\ a_0$. The string p is constructed from a set of symbols that a tree may contain, but there is no way to construct a tree from this string as the third children of the root (a_3) is missing.

The set of all ranked prefix notations over the ranked alphabet Σ is actually a context-free language generated by the context-free grammar

$$S \to a_3 SSS$$
$$S \to a_0$$

More generally, the set of all ranked prefix notations over any ranked alphabet $\mathcal{A}$ is a context-free language generated by a context-free grammar with rules of the form

$$S \to s_n S^n$$

where $n \in \mathbb{N}_0; s_n \in \mathcal{A}$. Strings from these languages can be recognized using a concept called *arity checksum* as described in [8].

Definition 19. Arity checksum *of a string* $w = a_1\ a_2\ \dots\ a_m, m \geq 1$ *over a ranked alphabet* $\mathcal{A}$*, denoted* $ac(w)$*, is defined as* $ac(w) = arity(a_1) + arity(a_2) + \dots + arity(a_m) - m + 1 = 1 - m + \Sigma_{i=1}^{m} arity(a_i)$.

Article [8] also shows (and gives a proof) that for any factor of the prefix notation of a tree T, the factor represents a subtree of T if and only if the arity checksum of the factor is zero and the arity checksum of any prefix of the factor not equal to the factor itself is greater or equal to one. The reasoning behind the arity checksum formula is that the sum of the arities is the number of all children. Therefore, the sum of arities plus one, to account for the root, must be equal to the number of nodes in the tree, which in turn must be equal to the length of the prefix notation.

$ac(w) = 1$	the root
$- m$	length of the prefix notation
$+ \Sigma_{i=1}^{m} arity(a_i)$	number of all children nodes

Also, if the checksum is equal to zero for any prefix of the input string, then the input string doesn't represent a single subtree, or the symbols are not ordered correctly. For example, consider the ranked alphabet $\mathcal{A} = \{a_0, a_2\}$ and the string $w \in \mathcal{A}^*, w = a_2\ a_0\ a_0\ a_1$. The arity checksum is $ac(w) = 1 - m + \Sigma i = 1^m arity(a_i) = 1 - 4 + 3 = 0$, but there is a prefix of w, $w' = a_2\ a_0\ a_0$ such that $ac(w') = 0$.

For bar notation, analogous checksum can be defined, the *bar checksum*. The problem of checking the prefix bar notation is the same as the balanced parentheses problem.

Definition 20. Bar checksum *of a string* $w = a_1\ a_2\ \ldots\ a_m$ *over a bar alphabet* $\mathcal{A}$, $] \in \mathcal{A}$*, denoted* $bc(w)$ *is defined as* $bc(w) = bc(a_1) + bc(a_2) + \ldots + bc(a_m) = \Sigma_{i=1}^{m} bc(a_i)$*, where:*

- $bc(]) = -1$ *and*
- $bc(a) = 1$ *for any* $a \in \mathcal{A},\ a \neq]$.

The bar checksum has the same properties with respect to subtrees and prefix bar notation as the arity checksum. A factor w of prefix bar notation of a tree represents its subtree if and only if bar checksum of w is equal to zero and the bar checksum of any prefix of w not equal to w is greater or equal to one.

Both checksums can be checked using a pushdown automaton. Since all the pushdown automaton does is counting, it only needs a single symbol in its stack alphabet and so it is a counter automaton. Unlike most automata, it does not accept a string by a state (it only needs a single state, which also happens to be final), but rather by an empty stack. For both notations, the stack operations are defined only by the input symbol and so in both cases the automaton is an input-driven pushdown automaton.

For arity checksum, the automaton starts with one symbol on the stack and then pops one symbol from the stack and pushes $arity(a)$ symbols on the stack for any input symbol a.

For bar checksum, the automaton starts with an empty stack, pushes a single symbol on stack for any input symbol different from the bar symbol and pops a single symbol from the stack for the bar input symbol. The simulation algorithm would have the same basic outline as algorithm 3 with appropriately modified arithmetical operations.

1.5 String searching

Stringology calls the searched string the *subject*, and the substring that is being search for the *pattern*. For the purposes of this discussion, let me denote the length of the pattern m and the length of the subject n.

Algorithm 3 Simulation of an arity checksum pushdown automaton

```
input: String inputString over a ranked alphabet
output: Boolean value indicating acceptance or rejection of the input string
function VerifyArityChecksum(inputString)
    counter ← 1
    for i ← 1 to ||inputString|| do
        if counter = 0 then
            return false
        end if
        counter ← counter − 1 + Arity(inputString[i])
    end for
    if counter = 0 then
        return true
    end if
    return false
end function
```

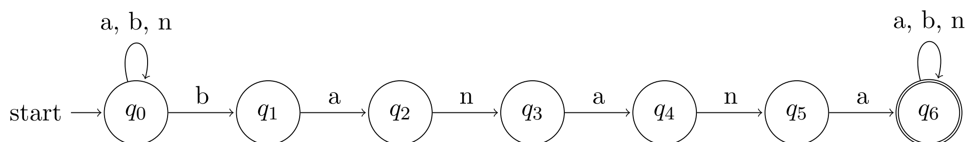


Figure 1.3: Nondeterministic pattern matching finite state automaton

In the next subsection, we are going to briefly describe pattern matching. We are doing this mostly only so that we can contrast pattern matching to indexing and it shouldn't be considered to be in any way an exhaustive discussion of the topic. The subsection 1.5.2 on page 11 is more important to the topic of this work.

1.5.1 Pattern matching

Pattern matching preprocesses the pattern to produce some kind of data structure that can be used to either accept or reject the subject. Typically, the preprocessing has $O(m)$ time complexity and the search $O(n)$. This is because the preprocessing usually involves one or more passes over the pattern, while the search iterates over the subject.

As a very simple example, I present a nondeterministic forward pattern matching finite automaton for exact string matching (see fig. 1.3) that accepts any subject string over the alphabet $\mathcal{A} = \{a, b, n\}$ containing the pattern $p = banana$. Table 1.1 then shows the trace of the automaton.

In practice, the language of all subjects containing a given pattern is not specified by a finite state automaton, but rather a regular expression.

Table 1.1: Trace of a nondeterministic forward pattern matching automaton

Input string	Symbol	Active states
$b\,a\,b\,a\,n\,a\,n\,a\,\$$		q_0
$a\,b\,a\,n\,a\,n\,a\,\$$	b	$q_0\ q_1$
$b\,a\,n\,a\,n\,a\,\$$	a	$q_0\ q_2$
$a\,n\,a\,n\,a\,\$$	b	$q_0\ q_1$
$n\,a\,n\,a\,\$$	a	$q_0\ q_2$
$a\,n\,a\,\$$	n	$q_0\ q_3$
$n\,a\,\$$	a	$q_0\ q_4$
$a\,\$$	n	$q_0\ q_5$
$\$$	a	$q_0\ \mathbf{q_6}$
ε	$\$$	subject accepted

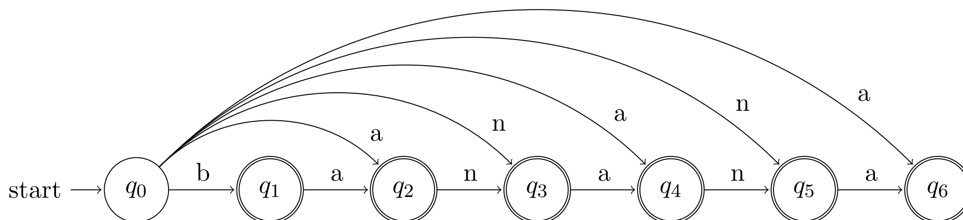


Figure 1.4: Factor automaton for the string "banana"

Provided, of course, that the language is regular. Many, if not most, programming environments contain built-in facilities for pattern matching using regular expressions, such as the `re` module in Python standard library, the `System.Text.RegularExpressions` namespace in the .NET framework class library, or the `java.util.regex` package in the Java Class Library.

These implementations, however, translate the regular expression to a DFA or a NFA and then simulate that automaton to actually perform the pattern matching. Algorithm 1.47 in [3] constructs a finite automaton equivalent to a given regular expression. Section 1.5 in [3] also gives complete formal definition of regular expressions.

1.5.2 Indexing

Whereas the pattern matching preprocesses the pattern, indexing preprocesses the subject, the result of which is surprisingly called the *index*. The preprocessing usually has linear time complexity in the length of the subject, $O(n)$, while the search has linear time complexity in the length of the pattern, $O(m)$. Presuming the subject is on average at least an order of magnitude longer than the pattern, indexing makes the preprocessing relatively expensive and search-

Table 1.2: Trace of a factor automaton

Input	Symbol	States
$a\,n\,a\,\$$		q_0
$n\,a\,\$$	a	$q_2,\ q_4,\ q_6$
$a\,\$$	n	$q_3,\ q_5$
$\$$	a	$q_4,\ q_6$
ε	$\$$	factor accepted, 2 occurrences found

Algorithm 4 Construction of nondeterministic factor automaton

```
input: A string over any alphabet
output: Nondeterministic factor automaton
q_0 ← new state
Q ← {q_0}
Σ ← ∅
Δ ← ∅
F ← ∅

for i ← 1 .. length of the input string do
    q_i ← new state
    Q ← Q ∪ {q_i}
    F ← F ∪ {q_i}
    s ← ith symbol in the input string
    Δ ← Δ ∪ {(q_{i−1}, s, q_i)}
    if i > 1 then
        Δ ← Δ ∪ {(q_0, s, q_i)}
    end if
end for

return (Q, Σ, Δ, q_0, F)
```

ing cheap compared to pattern matching. This makes indexing more suitable for applications where multiple queries are performed over large database and there are significantly more queries than changes to the database. Canonical examples include relational databases or world wide web search engines.

Factor automaton is a finite automaton representing the index of all factors of a string. Figure 1.4 gives example of a nondeterministic factor automaton for the string "*banana*". Table 1.2 then shows trace of the string "*ana*" being accepted by this automaton.

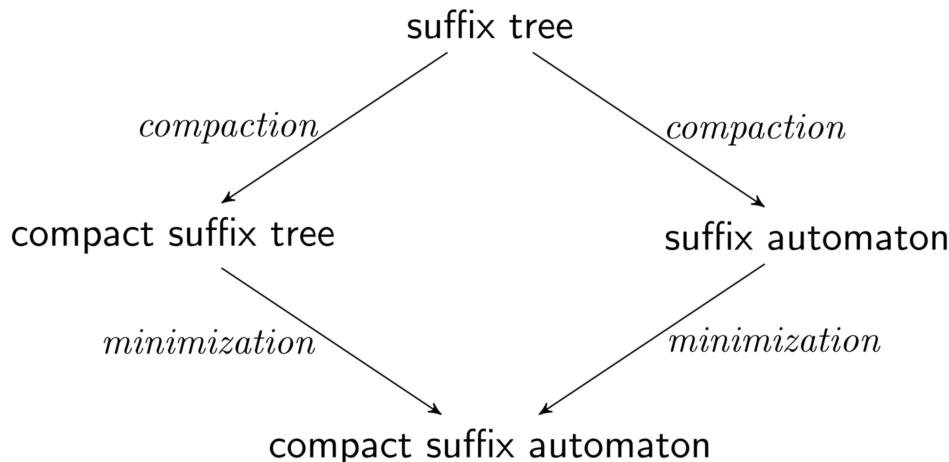


Figure 1.5: Conversions between suffix trees and automata

1.5.3 Suffix Automata

Definition 21. The suffix automaton *of a word x is the minimal deterministic automaton accepting Suff(x), the set of all suffixes of x. The suffix automaton of a word x is denoted $SA(x)$.*

Suffix automaton is an index of all suffixes of a string x. In literature, they are also known as Directed Acyclic Word Graphs, or DAWGs for short. They are related to factor automata in that in some cases (deterministic factor automaton vs. plain deterministic suffix automaton) they differ only in which states are marked as final.

Algorithm 5 constructs a suffix automaton on-line in linear time. It is a slightly rephrased version of the algorithm found in [4].

For the purposes of this thesis, deterministic compact suffix automata in particular are considered. This kind of automata are also known as Compact Directed Acylic Word Graphs, or CDAWGs.

Suffix automata are closely related to suffix trees in that if one views a suffix tree as an automaton, its minimization yields suffix automaton.

1.5.4 Position Heaps

Position heaps are relatively new structures that have been described, along with linear construction algorithm, in [7]. Like DAWGs, position heaps can generally index a set of strings and are not limited to set of all suffixes of a string, but this work uses the term "position heap of string x" to mean position heap of all suffixes of x.

Position heaps are structures that are, like suffix automata, related to suffix trees. Unlike suffix trees or suffix automata, however, they are not full

Algorithm 5 Construction of deterministic suffix automaton

```
input: A string x
output: Deterministic suffix automaton, DAWG
q0 ← new state
Q ← {q0}
Σ ← alphabet of x
δ ← ∅
F ← ∅

sink ← q0
q0.suffixLink ← nil

for i ← 1 to |x| do
    label ← x[i]
    newSink ← new state
    newSink.length ← sink.length + 1

    w ← sink
    while w ≠ nil and δ(w, label) ≠ nil do
        create edge from w to newSink labeled label
        w ← w.suffixLink
    end while

    if w = nil then
        newSink.suffixLink ← q0
    else
        v ← δ(w, label)
        if v.length = w.length + 1 then
            newSink.suffixLink ← v
        else
            newNode ← new state with the same outgoing edges as v
            newNode.length ← w.length + 1
            δ(w, label) ← newNode
            newSink.suffixLink ← newNode
            newNode.suffixLink ← v.suffixLink
            v.suffixLink ← newNode
            w ← w.suffixLink
            while w ≠ nil and v.length ≠ w.length + 1 do
                δ(w, label) ← newNode
                w ← w.suffixLink
            end while
        end if
    end if

    sink ← newSink
end for

while sink ≠ nil do
    F ← F ∪ {sink}
    sink ← sink.suffixLink
end while
```

Algorithm 6 Automaton Compaction

```
procedure CompactAutomaton(Q, Σ, δ, q0, F)
    stack ← empty stack
    visited ← ∅
    push q0 onto stack
    while stack is not empty do
        q ← pop state from stack
        if q ∉ visited then
            visited ← visited ∪ {q}
            for all (r, l1) such that δ(q, l1) = r do
                while IsReducible(q, F) do
                    l2 is a string and s a state such that δ(r, l2) = s
                    δ(q, l1) ← nil
                    l1 ← Concat(l1, l2)
                    δ(q, l1) ← s
                    Q ← Q\{r}
                    r ← s
                end while
                push r onto stack
            end for
        end if
    end while
end procedure

function IsReducible(q, F)
    return InDegree(q) = 1 and OutDegree(q) = 1 and q ∉ F
end function
```

indexes of all suffixes. Position heap for string x, denoted $PH(x)$, allows, based on some input string s, to select Occ, a subset of $Suff(x)$ containing at least all suffixes that can be written as sw in $O(||s|| + |Occ|)$ time. Their memory complexity is $O(||x||)$, but the number of nodes is exactly $||x||$. Thus, they can never have more nodes than corresponding suffix tree, but they also cannot have less nodes that corresponding compact suffix automaton.

Definition 22. *Let x be a string. $h(x)$ is the length of the longest substring y of x, which occurs at least $||y||$ times in x.*

As shown in [7], height of a position heap $PH(x)$ is at most $2h(x)$. In most texts, $h(x)$ can be expected to be significantly smaller than $||x||$.

Figure 1.6 shows the suffix tree for the string "banana" and the position heap for the same string for comparison. At first glance, it is clear there are similarities between the two structures. More on the relation between the suffix trees and position heaps in chapter 6 on page 49.

Algorithm 7 Searching for occurrences of patterns using a suffix automaton

input: A compact suffix automaton $SA(T)$ of text T and a pattern P
output: A set of tuples $(\mathit{offset}, \mathit{offset} + ||P||)$ representing $occ^T(P)$, the occurrences of P in T
function FindOccurrences($SA(T), P$)
  let *pref* be the longest prefix of P, which is represented by a state q in $SA(T)$
  let l be the label of the outgoing edge of q with the longest common prefix with $\mathit{pref}^{-1}P$
  let *lcp* be the longest common prefix of l and $\mathit{pref}^{-1}P$
  if $\mathit{lcp} < ||\mathit{pref}^{-1}P||$ **then**
    return $\emptyset$
  else if $\mathit{lcp} > 0$ **then**
    for each path of length *length* form state q through $\delta(q, l)$ to any final state **do**
      return $(||T|| - \mathit{length} - ||P||, ||T|| - \mathit{length})$
    end for
  else
    for each path of length *length* form state q to any final state **do**
      return $(||T|| - \mathit{length} - ||P||, ||T|| - \mathit{length})$
    end for
  end if
end function

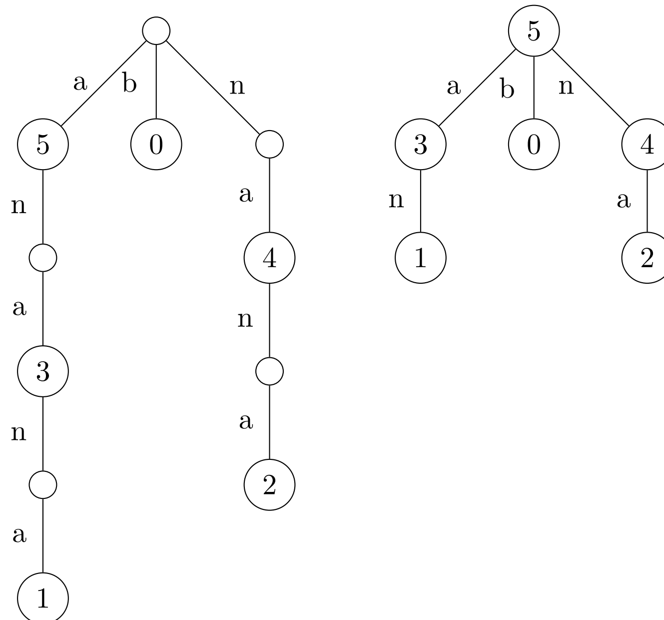


Figure 1.6: The suffix tree and the position heap for string “banana”

Algorithm 8 Searching for occurrences of patterns using a position heap

input: A position heap $PH(T)$ of text T and a pattern P
output: A set of tuples $(\mathit{offset}, \mathit{offset} + ||P||)$ representing some subset of $occ^T(P)$, the occurrences of P in T

```
function FindOccurrencesPH(PH(T), P)
    n ← root node of PH(T)
    result ← {(n.position, n.position + ||P||)}
    for i ← 1 to ||P|| do
        n ← child of n with edge labeled P[i]
        if n = nil then
            return result
        end if
        if ||T|| − n.position + 1 ≥ ||P|| then
            result ← result ∪ {n.position, n.position + ||P||}
        end if
    end for
    for each descendant d of n do
        if ||T|| − d.position + 1 ≥ ||P|| then
            result ← result ∪ {d.position, d.position + ||P||}
        end if
    end for
    return result
end function
```

1.6 Subtree patterns

In this text, a subtree pattern is defined as a ranked prefix notation of a subtree, which may contain $\mathcal{S}$-symbols. $\mathcal{S}$-symbol is a special symbol not in Σ, which matches any subtree. In order for the $\mathcal{S}$-symbol to satisfy the conditions for the arity checksum of prefix notation of a tree (see def. 19), its arity is defined to be 0.

Subtree patterns can be accepted by an input-driven pushdown automaton based on the factor automaton, but additional transitions have to be added to it to allow $\mathcal{S}$-symbols to be accepted. Figure 1.8 shows such automaton for the tree shown in figure 1.7.

Algorithm 9 constructs a tree pattern automaton for any tree in ranked prefix notation. The time complexity of the construction algorithm is $O(n)$ where n is the size of the tree for which the automaton is constructed. The time complexity of accepting a subtree in prefix notation by this automaton is $O(m)$ where m is the size of the accepted subtree.

In this text, I'll refer to the transitions of the form $(q_i, \mathcal{S}, S, q_j, \varepsilon)$ as "$\mathcal{S}$-*transitions*".

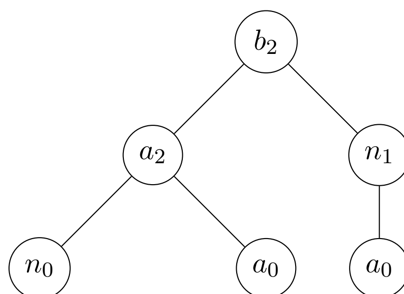


Figure 1.7: A ranked ordered tree

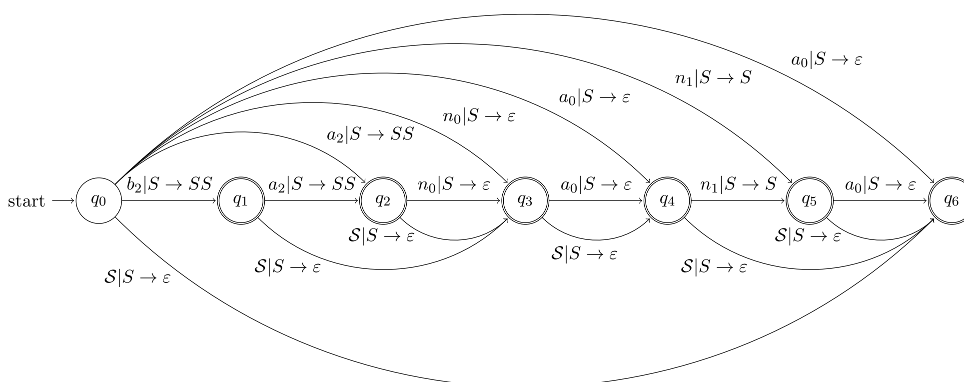


Figure 1.8: Nondeterministic tree pattern automaton for ranked prefix notation of the tree in fig. 1.7

Algorithm 9 Construction of nondeterministic tree pattern automaton

```
input: A tree in ranked prefix notation
output: Nondeterministic subtree and tree pattern automaton for the tree
q_0 ← new state
Q ← {q_0}
Σ ← ∅
Δ ← ∅
Z ← {S}
F ← ∅

for i ← 1 .. length of prefix notation do
    q_i ← new state
    Q ← Q ∪ {q_i}
    F ← F ∪ {q_i}
    s ← ith symbol in prefix notation
    Δ ← Δ ∪ {(q_{i−1}, s, S, q_i, S^{arity(s)−1})}
    if i > 1 then
        Δ ← Δ ∪ {(q_0, s, S, q_i, S^{arity(s)−1})}
    end if
end for

for i ← 1 .. length of prefix notation do
    d ← size of the subtree with the ith symbol as its root
    Δ ← Δ ∪ {(q_{i−1}, 𝒮, S, q_{i−1+d}, ε)}
end for

return (Q, Σ, Γ, Δ, q_0, Z, F)
```

CHAPTER 2

WBC bitmap index

2.1 Bit parallelism

Methods of simulating nondeterministic automata using bit parallelism encode the current configuration of the automaton as a bit vector and implement the transition operation as bitwise operations over two or more bit vectors.

Advantage of this approach is that the bitwise operation instructions in CPUs perform the operation (AND, OR, SHIFT) on all of the bits in a computer word in parallel. Bit parallelism is becoming increasingly more practical as the size of a computer word grows. Today's typical personal computer architectures use 32-bit or 64-bit words, which can already be used to simulate meaningful automatons in a single word.

There are several bit parallelism algorithms, namely Shift-Or, Shift-And or Shift Add and as discussed in [3], they have been shown to be applicable to several problems including approximate string matching as well as exact string matching.

To implement the subtree and tree pattern automaton described in section 1.6, modified Shift-And or Shift-Or algorithms could be used.

2.2 Shift-And algorithm

In the Shift-And algorithm, a bit vector $\vec{c}$ is used to represent the current configuration of the automaton. The active states are represented by the set bits. The original algorithm uses a bit vector array M of $|\Sigma|$ bit vectors of length n. Bit vectors are constructed so that $M[s][i] = 1$ iff $(q_i, s, q_{i+1}) \in \Delta$ for $s \in \Sigma$, $0 \leq i < n$. In this text and in the source code of my implementation, the bit vectors in M are referred to as "*transition masks*" or, simply, "*masks*".

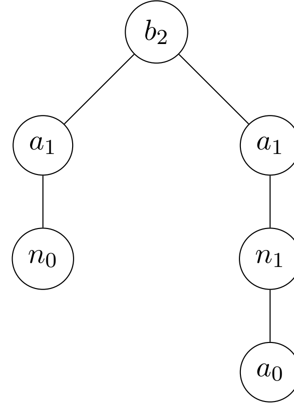


Figure 2.1: Another ranked ordered tree

The transition operation is then defined as

$$\vec{c} \leftarrow \begin{cases} M[s] & \text{for the first input symbol} \\ (\vec{c} \ll 1) \;\&\; M[s] & \text{for the subsequent input symbols} \end{cases}$$

for $s \neq \mathcal{S}$.

The initial state, q_0, is not represented by any bits since once the first symbol is accepted, q_0 cannot become active again. The operation for the first symbol simulates transitions from the initial state to states q_1, q_2, $\dots q_n$, whereas the operation for subsequent symbols simulates transitions of the form (q_i, s, q_{i+1}).

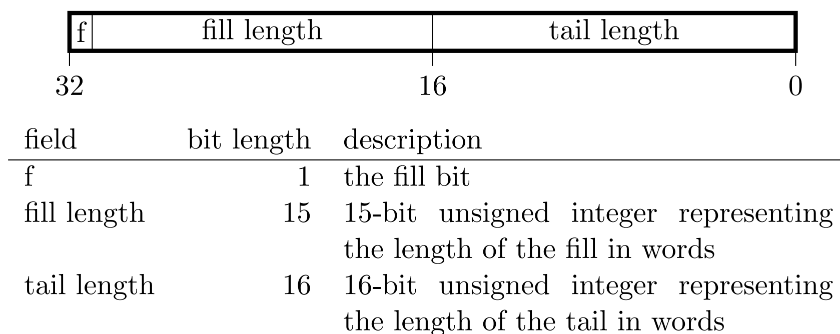
An extension to the original algorithm is needed to implement the $\mathcal{S}$-transitions. An integer array T is used such that $|T| = n$ and $T[i] = j$ for $1 \leq i, j \leq n$; $\delta(q_i, \mathcal{S}) = q_j$. In this text, the array T is referred to as "$\mathcal{S}$-*transition table*" or "*subtree transition table*".

The result of the transition operation for $s = \mathcal{S}$ is then defined as bit vector $\vec{d}$ such that $\vec{d}[j] = 1$ iff $\exists i$; $\vec{c}[i] = 1 \wedge T[i] = j$ for $1 \leq i \leq n$.

As [3] and [6] note, this algorithm has best performance when the bit vectors fit into a single computer word. With longer subjects, the algorithm has linear time complexity in n. Therefore, the obvious approach when trying improve upon it is to limit the number of bits (and possibly words) the algorithm has to operate on. In the next section, I'll describe one such approach and explain why I decided against using it for tree pattern automaton.

2.3 Compact representation

The main idea behind a nondeterministic automata compression method described in [6] is based on observation that the states in a nondeterministic suffix automaton can be split into contiguous groups such that each of these groups contains at most one active state at any time.



field	bit length	description
f	1	the fill bit
fill length	15	15-bit unsigned integer representing the length of the fill in words
tail length	16	16-bit unsigned integer representing the length of the tail in words

Figure 2.2: Layout of a WBC header

The basic Shift-And algorithm uses one bit for each symbol in subject. However, if the subject is factorized in such a way that each factor contains at most one occurrence of any symbol in Σ (so-called *minimal 1-factorization*), then one bit per such factor and the last accepted symbol are needed to encode the automaton's configuration. The bit says whether one of the symbols in the factor has been accepted and the last accepted symbol says which one.

Therein lies the problem with using this technique to simulate tree pattern automaton. The encoding cannot simply express the configuration of the automaton after accepting a $\mathcal{S}$-symbol. For example, consider the ranked ordered tree shown in figure 2.1. Its ranked prefix notation is "$b_2\, a_1\, n_0\, a_1\, n_1\, a_0$". Minimal 1-factorization of this string would split it into two factors, $f_1 = b_2\, a_1\, n_0$ and $f_2 = a_1\, n_1\, a_0$. The configuration of an automaton for this tree would be encoded as a tuple $(\vec{v}, s)$, where $\vec{v}$ is a 2-bit bit vector and s is a symbol in $\Sigma = \{a_0, a_1, b_2, n_0, n_1\}$.

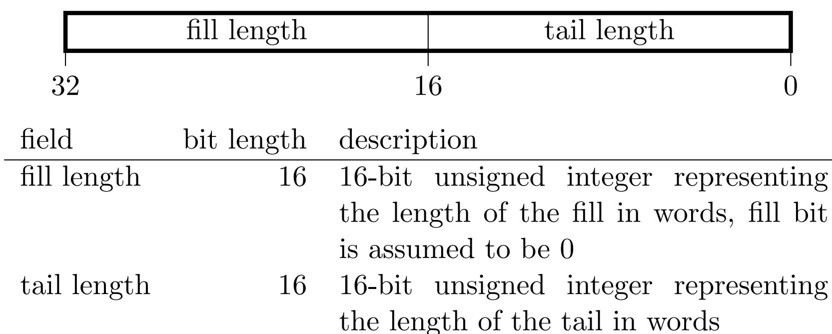
The tree pattern "$a_1\, \mathcal{S}$" matches two factors in this string, $a_1\, n_0$ and $a_1\, n_1\, a_0$. However, when accepting $\mathcal{S}$, there is no way to encode the new configuration, because $\mathcal{S} \notin \Sigma$.

2.4 Bit parallelism using run-length encoded bit vectors

The basic idea of this technique is to use run-length encoded bit vectors to save space and perform bitwise operations only over those words, that can actually have any bits set. First, I describe the encoding used and then I'll explain how it can be combined with the Shift-And algorithm.

2.4.1 Word-aligned bitmap encoding

Word-aligned Bitmap Coding (WBC) is a kind of run-length encoding described in [12] and [11]. WBC is a word-aligned version Byte-aligned Bitmap coding. It is designed in such a way so that bitwise operations on two en-



field	bit length	description
fill length	16	16-bit unsigned integer representing the length of the fill in words, fill bit is assumed to be 0
tail length	16	16-bit unsigned integer representing the length of the tail in words

Figure 2.3: Layout of adapted WBC header

coded bit vectors can be easily and efficiently implemented without sub-word operations. It is meant to be used to compress bitmap indices in relational databases and to support efficient queries on them.

A WBC encoded bit vector is a vector of words representing a series of so-called *runs*. Each run consists of a single word, the *header*, declaring the repeating bit, the number of repeating bits, the *fill*, and the number of words following the run, the *tail*. The length of the fill must be a multiple of the number of bits in a word and therefore it is expressed as number of words rather than number of bits. Layout of a WBC header word is shown in figure 2.2.

For the purposes of implementing the Shift-And algorithm using the WBC encoding, I have modified the encoding to only compress runs of zero bits. Because the fill bit can be assumed to be always 0, the fill bit doesn't need to be specified in the header word. The modified header word layout then looks as shown in figure 2.3. The reason for this that one or more 1-filled words in this application are just not very probable and assuming that the fill is always 0-filled greatly simplifies any operations over the encoded bit vectors. In the rest of the text, I'll use the term WBC to mean the modified version of WBC I have just presented.

Formally, the WBC encodes bit vector as n-tuple of 2-tuples $(f, \vec{t})$, where f is the length of the 0-fill in multiples of 32 such that $f \in \langle 0; 65535 \rangle$ and $\vec{t}$ is a vector of 32-bit bit vectors (32-bit machine is assumed). Offset o of the first bit in $\vec{t}$ of n-th run in the decoded bit vector is given as

$$o = 32 \times (f_n + \Sigma_{i=1}^{n-1} f_i + |\vec{t_i}|)$$

where f_i is the fill length of the i-th run and $\vec{t_i}$ is the tail of the i-th run.

To give an example, consider bit vector $\vec{v} = 0^{64}\, 1^{32}\, 0^{32}\, 1^{32}$. This bit vector could be encoded as $((2, (1^{32})), (1, (1^{32})))$. The coding doesn't require any repeating 0-bits that could be encoded as a fill to actually be encoded as such. Actual implementations may fold repeating 0-bits into a fill only if

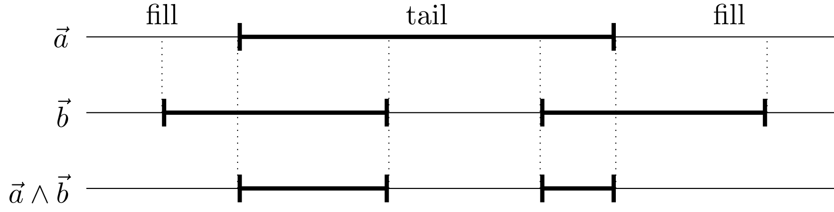


Figure 2.4: Bitwise AND of two WBC bit vectors

their number is higher than a certain limit. Therefore, $((2,(1^{32},0^{32},1^{32})))$ or $((0,(0^{32},0^{32},1^{32},0^{32},1^{32})))$ are valid WBC encoding of $\vec{v}$ too.

2.4.2 Shift-And algorithm using WBC encoded bit vectors

Algorithm 10 Bitwise AND over two word vectors

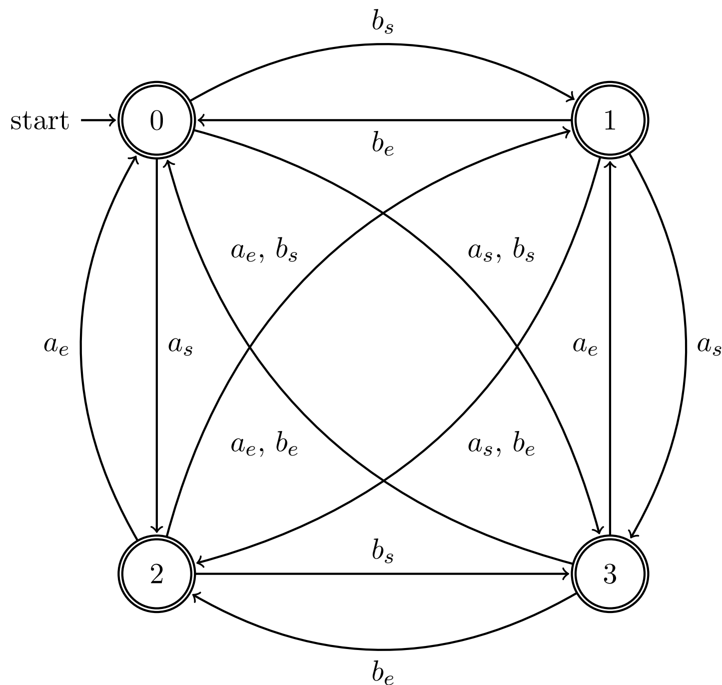
```
procedure AND(a, b, length)
    c ← new word interval of length length
    for i ← 0, length − 1 do
        c[i] = a[i] ∧ b[i]
    end for
    return c
end procedure
```

The WBC encoding has been designed to easily support bitwise AND, OR and other related operations, but not the bit shift operation. However, since the Shift-And algorithm only shifts by a single bit, any individual bit can only cross the word boundary from the most significant bit of a word to the least significant bit of the following word. This means the Shift-And operation can be easily implemented by slightly extending the AND operation.

For AND operation over two bit vectors, $\vec{a}$ and $\vec{b}$, any word in the resulting bit vector $\vec{a} \wedge \vec{b}$ can have bits set only if words at offsets that happen be within a tail in both of the input bit vectors. As a result, the tails on the resulting bit vector are the intersection of the tails in the input vectors (see figure 2.4).

The resulting vector can be calculated in linear time by iterating over the runs in $\vec{a}$ and $\vec{b}$ and selecting the appropriate action to be performed depending on whether the current offset is inside a tail in $\vec{a}$ and whether it is inside a tail in $\vec{b}$. See algorithm 11 for more detailed description. The algorithm basically simulates the finite state automaton shown in figure 2.5.

The bit shift is then done by shifting the individual words and keeping track of the carry bit (the bit that crosses the word boundaries). In some cases, the carry bit would shift into a word that wouldn't be included in any tail in the simple And algorithm. That is why the tails must be extended



symbol	meaning
a_s	start of a tail in $\vec{a}$
a_e	end of a tail in $\vec{a}$
b_s	start of a tail in $\vec{b}$
b_e	end of a tail in $\vec{b}$

Figure 2.5: Run intersection finite automaton

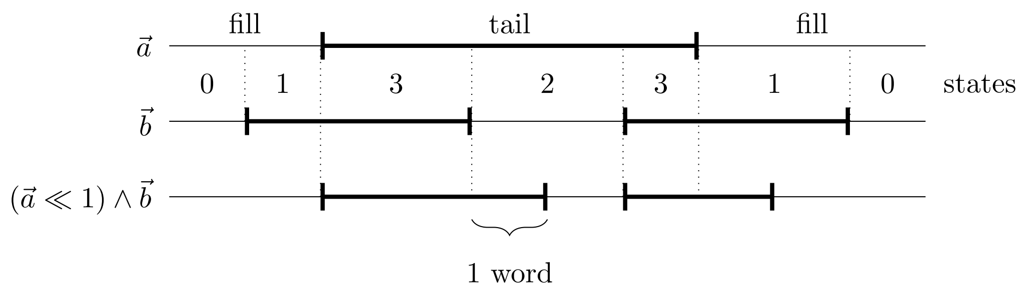


Figure 2.6: Shift-And for two WBC bit vectors

Algorithm 11 Bitwise AND over two WBC bit vectors

input: WBC encoded bit vectors $\vec{a}$ and $\vec{b}$
output: WBC encoded bit vector $\vec{c}$; $\vec{c} = \vec{a} \wedge \vec{b}$
require: $\vec{a}$ has the same bit length as $\vec{b}$

result $\leftarrow$ new empty bit vector

$i_a \leftarrow$ run iterator for $\vec{a}$
$i_b \leftarrow$ run iterator for $\vec{b}$
while i_a is not at the end of $\vec{a}$ **do**
  $d \leftarrow$ distance to closest beggining or end of a tail in either $\vec{a}$ or $\vec{b}$
  if i_a is a tail $\wedge$ i_b is a tail **then**
    add $(i_a \ll 1) \wedge i_b$ to result
  else if $i_a[-1]$ is in a tail $\wedge$ i_b is a tail **then**
    add $HSB(i_a[-1]) \wedge i_b[0]$ to previous tail
    add 0-fill of length $d - 1$ to result
  else
    add 0-fill of length d to result
  end if
  move i_a and i_b by d
  $run_a \leftarrow$ next run in $\vec{a}$ **if** at end of run_a
  $run_b \leftarrow$ next run in $\vec{b}$ **if** at end of run_b
end while

by one word in some cases as shown in figure 2.6. This figure also shows the states the run intersection algorithm passes through as it scans the two bit vectors.

CHAPTER 3

Full and linear index

This data structure and the accompanying search algorithm have been first described in [9]. In this work, it will be abbreviated as *FLLI*.

For an indexed tree T, the data structure consists of three parts:

- **compact suffix automaton** $SA(pref(T))$ (can be constructed by algorithms 5 and 6, although direct on-line construction algorithms for compact suffix automatons exist),
- **subtree jump table** $SJT(T)$ as defined by algorithm 12
- and a temporary *reverse array* of length $||pref(T)||$ denoted here $Rev^{length}_{occurrences}$.

Algorithm 12 Construction of subtree jump table

```
function ConstructSJT(T)
    SJT ← new array of length ||pref(T)||
    ConstructPartialSJT(T, 1, SJT)
    return SJT
end function

function ConstructPartialSJT(T, rootIndex, SJT)
    index ← rootIndex + 1
    for i ← 1 to pref(T)[rootIndex].arity do
        index ← ConstructPartialSJT(T, index, SJT)
    end for
    SJT[rootIndex] ← index
    return index
end function
```

It is interesting to note that the *subtree jump table* structure is almost identical to what has been termed *subtree transition table* in the WBC index. The only difference is that indices in SJT point to symbol immediately *after* the prefix notation of the corresponding subtree, while STT points to the last symbol of the same subtree.

Like the WBC bitmap index, this index exploits ranked prefix notation of a tree and transforms the arbological problem of searching for all occurrences of a subtree to stringological problem of searching for all occurrences of a factor. Full and linear index of a tree uses a compact suffix as solution to the problem of searching for exact tree patterns (strings over the alphabet of $pref(T)$).

The only problem arises with accepting *subtree patterns* and handling $\mathcal{S}$-symbols in input. As with the bitmap index, $\mathcal{S}$-symbols inherently introduce nondeterminism, because in each occurrence of the pattern, an $\mathcal{S}$-symbol may match a factor of different length. In other words, adding additional edges to $SA(pref(T))$ that would allow the automaton to accept $\mathcal{S}$-symbols would also make the the automaton nondeterministic.

3.1 Search Algorithm

The prefix notation $pref(P)$ of any subtree pattern P can be written as:

$$pref(P) = P_1\,\mathcal{S}\,P_2\,\mathcal{S}\,\ldots\,\mathcal{S}\,P_k, k \geq 1$$

Definition 23. *Let* $pref(P) = P_1\,\mathcal{S}\,P_2\,\mathcal{S}\,\ldots\,\mathcal{S}\,P_k$ *be the prefix notation of a subtree pattern* P *over alphabet* $A \cup \{\mathcal{S}\}$*, where no factor* P_i*,* $1 \geq i \geq k$ *contains symbol* $\mathcal{S}$*. The factor* P_i *is called* i-th subpattern *of* P*.*

Then the outline of the algorithm is.

Algorithm 13 Outline of the search algorithm

function SearchFLLI(T, P)
  for each P_i **do**
    1. accept supattern P_i using compact suffix automaton $SA(pref(T))$
    2. use $SJT(T)$, the *subtree jump table*, to accept the $\mathcal{S}$ symbol following P_i if there is any
    3. merge occurrences of $P_1\,\mathcal{S}\,P_2\,\ldots\,\mathcal{S}\,P_{i-1}\,\mathcal{S}$ from the previous iteration and occurrences of $P_i\,\mathcal{S}$ computed in the previous step using the $Rev^{||pref(T)||}_{occ(P_1 \ldots P_{i-1}\,\mathcal{S})}$ array
  end for
end function

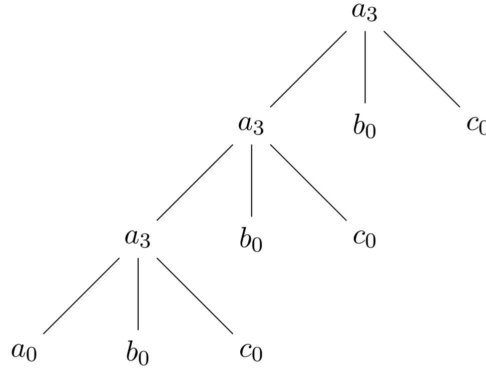


Figure 3.1: Ranked ordered tree T_1 from example 1

The algorithm works in $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|)$ time, where $occ(P_i)$ is the set of all occurrences of subpattern P_i in T.

However, mostly for implementation reasons, it is beneficial to use slightly different form of $pref(P)$, which groups subsequent $\mathcal{S}$ symbols.

$$pref(P) = P_1\, G_1\, P_2\, G_2\, \ldots\, G_{k-1}\, P_k, k \geq 1$$

Definition 24. *Let $pref(P) = P_1\, G_1\, P_2\, G_2\, P_3\, \ldots\, G_{k-1}\, P_k$ be the prefix notation of a subtree pattern P over alphabet $A \cup \{\mathcal{S}\}$, where no factor P_i, $1 \geq i \geq k$ contains symbol $\mathcal{S}$ and each factor G_i is a string such that $G_i \in \{\mathcal{S}\}^* \backslash \{\varepsilon\}$. The factor G_i is called i-*th $\mathcal{S}$-symbol group of P.

This allows the algorithm to be implemented slightly more efficiently and makes handling empty subpatterns P_i easier, because only the first or last subpatterns (P_1 and P_k respectively) can be empty. If any other subpattern P_i was empty, string $G_{i-1}\, P_i\, G_i$ would belong to language $\{\mathcal{S}\}^* \backslash \{\varepsilon\}$ and thus, by the definition 24, could actually be a single $\mathcal{S}$-symbol group on its own.

The algorithm 14 describes the search function SEARCHFLLI in little more detail. The VERIFYARITYCHECKSUM function is described in algorithm 3. The MERGEOCCURENCES function is the same as the one given in [9].

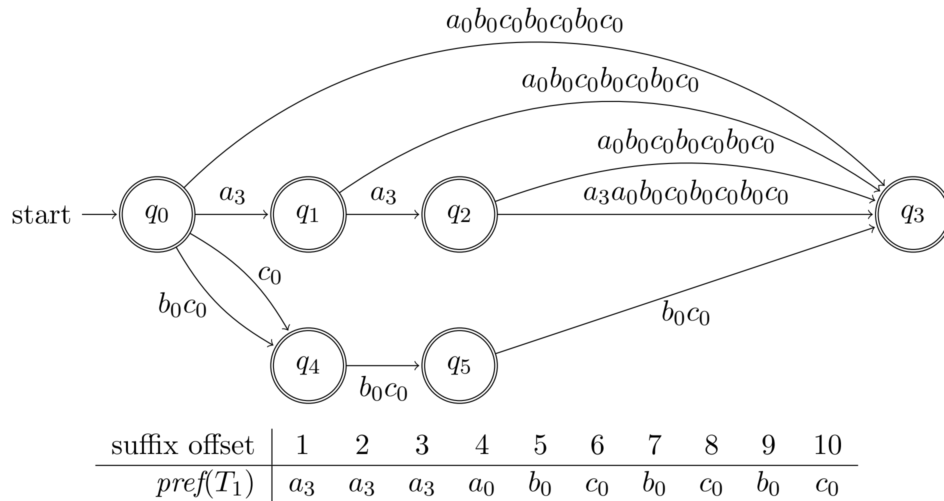
Example 1 shows a ranked ordered tree, which used in example 2 to demonstrate how the FLLI search algorithm works.

Example 1. *Consider ranked ordered tree T_1 over $\mathcal{A} = \{a_3, b_0, c_0\}$, such that $pref(T_1) = a_3\, a_3\, a_3\, a_0\, b_0\, c_0\, b_0\, c_0\, b_0\, c_0$.*

Example 2. *Consider the tree T_1 from example 1 and a subtree pattern $P, pref(P) = a_3\, \mathcal{S}\, b_0\, c_0$.*

Algorithm 14 Full and linear index search algorithm

```
function SearchFLLI(T, P)
    if pref(P) = S then
        return set of all subtrees of T
    end if
    if not VerifyArityChecksum(pref(P)) then
        error – invalid pattern
    end if
    prevOcc ← FindOccurrences(SA(pref(T)), P_1)
    for i ← 2 to k do
        for each occurrence (first, last) ∈ prevOcc do
            for j ← 1 to ||G_{i−1}|| do
                (first, last) ← (first, SJT(T)[last])
            end for
        end for
        if P_i ≠ ε then
            nextOcc ← FindOccurrences(SA(pref(T)), P_i)
            prevOcc ← MergeOccurences(prevOcc, nextOcc)
        end if
    end for
    return prevOcc
end function
```



suffix offset	1	2	3	4	5	6	7	8	9	10
pref(T_1)	a_3	a_3	a_3	a_0	b_0	c_0	b_0	c_0	b_0	c_0

Figure 3.2: Ranked prefix notation *pref*(T_1) and compact suffix automaton $SA(pref(T_1))$ for tree T_1 from example 1

$pref(T_1)$	a_3	a_3	a_3	a_0	b_0	c_0	b_0	c_0	b_0	c_0
suffix offset i	1	2	3	4	5	6	7	8	9	10
$SJT[i]$	11	9	7	5	6	7	8	9	10	11

Figure 3.3: Subtree jump table SJT for FLLI index of tree T_1 from example 1

Pattern P_1 can be written as $P = P_1 G_2 P_2$, where

$$P_1 = a_3,$$
$$G_2 = \mathcal{S}$$
$$\text{and } P_2 = b_0\, c_0.$$

1. *First step in the FLLI search phase (the* SearchFLLI *algorithm) is to search for all occurrences of subpattern $P_1 = a_3$ in $pref(T_1)$ using the compact suffix automaton $SA(pref(T_1))$ shown in figure 3.2 and the algorithm 7. The final configuration of the automaton would end in state q_1, from which there are 3 paths leading to final states labeled $a_0\, b_0\, c_0\, b_0\, c_0\, b_0\, c_0$, $a_3\, a_0\, b_0\, c_0\, b_0\, c_0\, b_0\, c_0$ and $a_3\, a_3\, a_0\, b_0\, c_0\, b_0\, c_0\, b_0\, c_0$. This yields occurrences*

$$occ(P_1) = \{(1,2),(2,3),(3,4)\}.$$

2. *Second step is to use SJT table to compute $occ(P_1\mathcal{S})$ from $occ(P_1)$.*

$$\begin{aligned} occ(P_1\mathcal{S}) &= \{(1, SJT[2]),(2, SJT[3]),(3, SJT[4])\} \\ &= \{(1,9),(2,7),(3,5)\} \end{aligned}$$

3. *Third step is to compute occurrences of the second subpattern $P_2 = b_0 c_0$:*

$$occ(P_2) = \{(5,7),(7,9),(9,11)\}.$$

4. *As a final step in this example, the* MergeOccurences *algorithm is used to merge $occ(P_1\mathcal{S})$ and $occ(P_2)$ calculated in the previous steps into $occ(P_1\mathcal{S}P_2) = occ(P)$. This step itself consists of two sub–steps:*

   a) *The $Rev_{\{(1,9),(2,7),(3,5)\}}$ array is calculated:*

i	1	2	3	4	5	6	7	8	9	10
$Rev_{\{(1,9),(2,7),(3,5)\}}[i]$	-1	-1	-1	-1	3	-1	2	-1	1	-1

   b) *Merge is performed using the Rev array:*

$$occ(P_1\mathcal{S}) = (1, 9), (2, 7), (3, 5)$$

$$Rev_{\{\dots\}} = $$

$Rev[i]$	-1	-1	-1	-1	3	-1	2	-1	1	-1
i	1	2	3	4	5	6	7	8	9	10

$$occ(P_2) = (5, 7), (7, 9), (9, 11)$$

$$occ(P_1 G_2 P_2) = (3, 7), (2, 9), (1, 11)$$

CHAPTER 4

Position heap index

The position heap index is rather simple modification of the full and linear index. The compact suffix automaton $SA(\mathit{pref}(T))$ is simply replaced by position heap $PH(\mathit{pref}(T))$.

Like the FLLI search algorithm, the position heap index search has $\mathcal{O}(m + \sum_{i=1}^{k} |occ(P_i)|)$ time. This is because the position heap has the same asymptotic bounds for searching for l occurrences of pattern P in text T as the suffix automaton, that is $\mathcal{O}(||P|| + l)$. In practice, however, the height of the position heap, which is at most $2h(T)$, can be significantly smaller than $||P||$ and so will be the search time for position heap as compared to suffix automaton.

Searching for occurrences of subpatterns using position heap, however, will have false positives. We predicted this will not be a big problem, since in order to have a false positive for the occurrence of the string $P_i \, \mathcal{S} \, P_{i+1}$, there must be an occurrence $(\mathit{first}_i, \mathit{last}_i)$ of P_i and an occurrence $(\mathit{first}_{i+1}, \mathit{last}_{i+1})$ of P_{i+1} such that $SJT[\mathit{last}_i] = \mathit{first}_{i+1}$. In order for a false positive to end up in the final search result, this coincidence must happen for each pair of subpatterns P_i and P_{i+1} in the subtree pattern P.

The probability of such coincidence seems to be relatively low. However, once the index had been implemented, the number of false positives in the search results were measured and the results are given in chapter 7.

Example 3. *Consider the tree T_1 from example 1 and a subtree pattern $P, \mathit{pref}(P) = a_3 \, \mathcal{S} \, b_0 \, c_0$ from example 2.*

Position heap for $\mathit{pref}(T_1)$, $PH(\mathit{pref}(T_1))$ is shown in figure 4.1. This position heap is used in this example. Note this position heap has considerably more states than the compact suffix automaton for $\mathit{pref}(T_1)$, but in general, position heaps will need to consider fewer symbols of the input string than an equivalent compact suffix automaton.

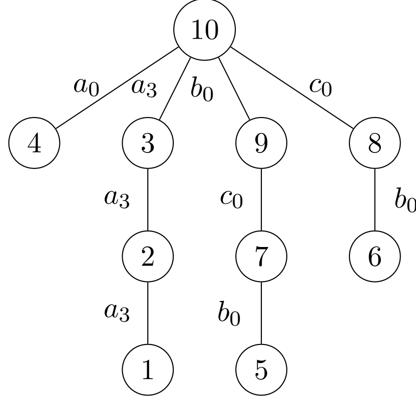


Figure 4.1: Position heap $PH(T_1)$ for tree T_1 from example 1

The search algorithm of the position heap index then proceeds in a very fashion very similar to FLLI (see example 2), with the exception that it uses the FindOccurrencesPH *function (algorithm 8) instead of* FindOccurrences *(algorithm 7).*

1. *Compute the approximate occurrences of subpattern P_1. Note that there is one false positive $(10, 11)$. This part of normal function of the position heap. In fact, occurrence $(10, 11)$ will be returned by any query using the position heap in figure 4.1, because it is represented by the heap's root node.*

   $\text{FindOccurrencesPH}(\mathit{PH}(\mathit{pref}(T_1)), \mathit{pref}(P_1)) = \{(10, 11), (3, 4), (2, 3), (1, 2)\}$

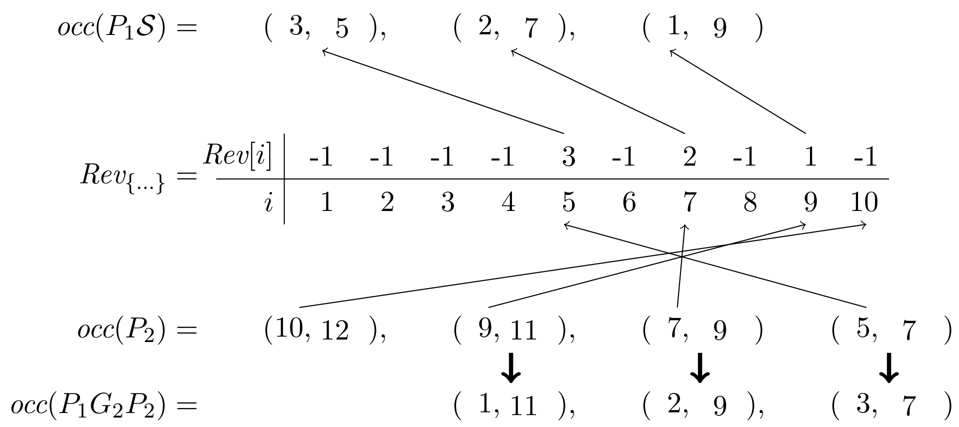
2. *Second step is to use SJT table to compute $occ(P_1\mathcal{S})$ from $occ(P_1)$. Note that occurrence $(10, 11)$ is eliminated, because $SJT[11]$ is undefined.*

$$\begin{aligned} occ(P_1\mathcal{S}) &= \{(3, SJT[5]), (2, SJT[3]), (1, SJT[9])\} \\ &= \{(3, 5), (2, 7), (1, 9)\} \end{aligned}$$

3. *Third step is to compute occurrences of the second subpattern $P_2 = b_0c_0$:*

   $\text{FindOccurrencesPH}(\mathit{PH}(\mathit{pref}(T_1)), \mathit{pref}(P_2)) = \{(10, 12), (9, 11), (7, 9), (5, 7)\}.$

4. *The final step is the same as with the FLLI search algorithm:*



occ(P1S) =
(3, 5), (2, 7), (1, 9)
Rev{...} =
Rev[i] -1 -1 -1 -1 3 -1 2 -1 1 -1
i 1 2 3 4 5 6 7 8 9 10
occ(P2) =
(10, 12), (9, 11), (7, 9) (5, 7)
occ(P1G2P2) =
(1, 11), (2, 9), (3, 7)

CHAPTER 5

Implementation

5.1 Basic architecture

My implementation consist of two main modules, each with its own executable. Both `indexer` and `lindexer` read an XML file, index it and save the index in a binary form to a file. The difference is `indexer` produces WBC and position heap indexes as described in this work, while `lindexer` produces full linear index as described in [?]. Subsequently, the `subtree` reads the binary index, reads a pattern from another XML file, performs the search and prints out the results. It is the `subtree` module that actually simulates the nondeterministic pushdown automaton.

5.2 Tools used

For the sake of performance, I have decided to implement the automaton simulator in C++. At first, I thought a practical indexer could be written in Python. I did write a prototype version in Python. One of the advantages was that it was easy to add a new frontend for different tree format, such as XML or JSON. This prototype version also used a DOM XML parser. This solution quickly proved to be unusable as it ran out of memory and failed for XML files with just couple of tens of megabytes in size. It was obvious that using a more memory efficient language and a stream-oriented XML parser was needed.

Rewriting the indexer to C++ using the Expat XML parser[1] limited it to XML input only, but brought significant increases in performance and stability. Expat parser, though not exactly conforming to the SAX API, uses the same general idea and basically offers prefix bar notation view of the XML

[1] `http://expat.sourceforge.net`

document. For more details on Expat, see section 5.2.1. Since my solution uses ranked prefix notation rather then prefix bar notation for subtree matching, my indexer employs algorithm 2 to convert the prefix bar notation to ranked prefix notation.

Both the indexer and subtree matcher use only statically linked libraries, with Expat parser library the only third-party library used. Both use STL containers wherever possible and appropriate. In particular, std::vector is used heavily, mostly to implement the bit vectors.

Classes defined in the <iostream> header file are used for IO, both writing and reading the binary index and text output.

To compile the C++ sources, I have used GCC in combination with Make. To debug the binaries, I have used GDB and Valgrind. My editor of choice was Vim and for version control I used Git.

5.2.1 Expat parser

The basic usage of the Expat parser is as follows:

1. initialize the parser,
2. set the event handlers and the user data pointer,
3. feed the parser with chunks of the XML file character data,
4. and finally, release the resources used by the parser.

```
XML_Parser XML_ParserCreate(const XML_Char* encoding)
```

Creates and initializes a new parser.

```
void XML_SetElementHandler(XML_Parser p,
                           XML_StartElementHandler start,
                           XML_EndElementHandler end)
```

The Expat parser keeps pointers to functions which it calls when an event, such as the beginning or the end of an element, occurs. This function sets the pointers to functions handling these two events. There are other handlers including handlers for various error states, but these two handlers are the two necessary to read the tree structure of the document.

```
void XML_SetUserData(XML_Parser p, void* userData)
```

Expat is a C library, not C++. In order for the client to be able to keep state between handler calls, the parser passes a pointer to all of the handlers. This pointer is set by the user a may point to any kind of user data including C++ class instance.

```
int XML_Parse(XML_Parser parser,
              const char* s,
              int len,
              int isFinal)
```

Parses a chunk of the input. This function is meant to be called repeatedly on chunks of character data until the entire XML file has been fed to the parser. This minimizes the number of IO operations and allows the program to be written in such a way that it doesn't need to dynamically allocate memory during parsing.

```
void XML_ParserFree(XML_Parser parser)
```

Frees memory used by the parser.

5.3 WBC index format

Index consists of the following parts:

- *Symbol table* is a list of all unique ranked symbols present in the tree. This table is used to translate ranked symbols to integer indices which are used to look up transition masks in the transition table (see below).
- *Subtree transition table* ($\mathcal{S}$-transition table) contains, for each symbol in the prefix notation of the tree, the index of the last symbol of the subtree that begins by symbol immediately adjacent to it. This table is used to implement the state transitions taken when accepting $\mathcal{S}$ symbol (see section 1.6 on page 17).
- *Origins table*, contains one string per each symbol in the prefix notation of the tree. This table is used to translate to search results to information about the location of a given tree node in the data source. My implementation of the indexer stores the line numbers of the opening and closing tag of the corresponding XML element here. The line numbers are given as `sed`[2] commands.
- *Transition mask table*, is a list of transition masks used in the implementation of the shift-and operation. The transition mask table contains a mask for each unique ranked symbol. A ranked symbol's transition mask has the same index in the mask table as the ranked symbol has in the symbol table.

Table 5.1 shows the primitive data types used in the binary format of the index, shown in table 5.5. To store strings, I have decided to use Pascal strings (see table 5.2) rather than null terminated C-strings. This is to keep the number of IO operations to minimum.

[2] `sed` (short for stream editor) is a Unix utility used to apply transformations to a text.

Table 5.1: Primitive data types

Name	Size (B)	Description
`uint8`	1	8-bit unsigned integer
`uint32`	4	32-bit unsigned integer in network byte order (big endian)
`char`	1	8-bit ASCII character

Table 5.2: `PString` data structure

Type	Interpretation
`uint8`	Length of the string
`char[length]`	ASCII string

Table 5.3: `RankedSymbol` data structure

Type	Interpretation
`PString`	Label
`uint32`	Arity, rank

5.4 Bit vector representation

There are two classes that represent bit vectors. `BitVector` (see fig. 5.1) is a simple implementation using vector of 32-bit unsigned integers as its backing store. `WBCBitVector` (see fig. 5.2) also uses a 32-bit unsigned integer vector as its backing store, but the bit vector is encoded using WBC and the classes provides a run iterator which can be used for iterating over the individual runs and reading the literal words in run tails.

For performance reasons, I tried to avoid dynamic memory allocation as much as possible. This meant I wanted to implement both Shift-And algorithm and WBC algorithm in-place. While this is trivial for regular bit vector, it would be very difficult, if not impossible, with two WBC vectors as offsets and lengths of the resulting runs may be different from the source vectors.

For this reason, I decided that unlike the transition masks, the current automaton configuration would be stored in a `BitVector` instance, which would also maintain a list of word intervals that are "inside" a tail (the `BitVector::activeIntervals_` class member). This way any word read from the bit vector would be written back at the same offset, while the run intersection algorithm can be still implemented in the same way.

5.5 Factor automaton simulation

The basic outline of the simulation algorithm is this:

1. read the index,

Table 5.4: `WBCBitVector` data structure

Type	Interpretation
`uint32`	WBC bitvector bit size
`uint32`	WBC bitvector run count
`uint32`	WBC bitvector word size
`uint32[]`	words

Table 5.5: Index file layout

Type	Interpretation
`char[4] "INDX"`	Index sigil for error checking and debugging purposes
`char[4] "NAME"`	Symbol table sigil
`uint32`	Symbol table size
`RankedSymbol[symTableSize]`	Symbol table items (see tbl. 5.3)
`char[4] "SJMP"`	$\mathcal{S}$-transition table sigil
`uint32`	$\mathcal{S}$-transition table size
`uint32[sTableSize]`	$\mathcal{S}$-transition table items
`char[4] "ORIG"`	Origins table sigil
`uint32`	Origins table size
`PString[]`	Origins table items (see tbl. 5.2)
`char[4] "MSKW"`	Transition mask table sigil
`WBCBitVector[]`	Transition masks items (see tbl. 5.4)

2. read the pattern and use the symbol table to translate its symbols,
3. set the current state to the transition mask of the first pattern symbol,
4. for each of the rest of the pattern symbols, perform the Shift-And operation on the current state and the transition mask for the current pattern symbol

5.6 Full and linear index format

This type of index uses the same format for *symbol table*, `subtree transition table` (or `subtree jump table`, as it is called [9]), and *origins table* as the WBC index. These parts of the index are event serialized in the same way.

The main difference in the format is that rather than the transition masks, this index must contain suffix automaton. Both adjacency list and transition matrix implementations have been tried and for this application, adjacency list was found to have better performance by an order of magnitude.

```
struct WordInterval {
        int offset;
        int length;
};

class BitVector {
private:
        int                  size_; // size in bits
        vector<uint32_t>     words_;
        vector<WordInterval> activeIntervals_; // imitates runs

public:
        // used in regular Shift-And
        void BitVector::shiftLeftAnd(const BitVector& other);

        // used in WBC Shift-And, uses activeIntervals_ member
        void BitVector::shiftLeftAnd(const WBCBitVector& other);

        class WordIterator {
        public:
                WordIterator(const BitVector& vector);

                vector<uint32_t>::iterator word;
                vector<uint32_t>::iterator end;
                int                        fill;

                inline int tail() const;
                inline bool next();
        };
};
```

Figure 5.1: Abbreviated declaration of the `BitVector` class

In either case, both in-memory representations of the suffix automaton are serialized the same way and command line options `-l` and `-t` can be used to switch between the adjacency list and transition matrix implementations respectively.

For performance reasons, the edge labels are not represented as strings, but rather as tuple (*offset*, *length*), which is implemented by the structure `Factor` (see fig. 5.4) in the code. This structure also contains `longestCommonPrefix` method, which can be used to determine longest common prefix of either two factors of the same string (i.e. their `start` and `length` members are indexes into the same string, used while building the suffix automaton, for example), or two factors of two different strings (used when simulating the suffix automaton to accept subtree pattern).

Suffix automaton itself is represented by the `SuffixAutomaton` class. This class simply contains a list of all states. More than its data are its methods,

```
class WBCBitVector {
private:
        int               size_; // size in bits
        vector<uint32_t> words_;

public:
        class WordIterator {
        public:
                WordIterator(const WBCBitVector& vector);

                vector<uint32_t>::const_iterator word;
                vector<uint32_t>::const_iterator end;
                int                              fill;

                inline int tail() const;
                inline bool next();
        };
};
```

Figure 5.2: Abbreviated declaration of the `WBCBitVector` class

```
typedef uint32_t PreorderIndex;

struct Factor {
        PreorderIndex start;
        PreorderIndex length;
};
```

Figure 5.3: Abbreviated declaration of the `Factor` structure

namely `slowFind` overloaded method, which can be used to find all occurrences of a factor in the indexed string.

To actually construct the automaton, the `SAConstructor` class is used.

5.7 Position heap index format

Format of the position heap index is almost identical to the format of the FLLI index. The only difference is that the serialized index uses different sigil (magic bytes): `PNDX`.

During runtime, the index is represented by the `PHIndex` class, whose implementation differs from `LIndex` mainly in using different search method, which interprets that graph structure associated with the index as position heap rather than compact suffix automaton.

```
struct SAState {
        bool             isFinal;
        PreorderIndex    length;
        SAState         *suffixLink;
        vector<SAEdge*>  edges;
};

struct SAEdge {
        Factor   label;
        SAState *target;
};

class SuffixAutomaton {
private:
        vector<SAState*> states;
};
```

Figure 5.4: Abbreviated declaration of the `Factor` structure

Except for that, even the position heap uses the same serialization format as the compact suffix automaton in FLLI with the suffix automaton's *length* state attributes interpreted as position heap's *position* attributes.

5.8 Compact suffix automaton simulation

We have created two implementations of finite automata: adjacency list implementation represented by the `SuffixAutomaton` class and transition matrix implementation represented by the `Automaton` class.

Both implementations are serialized the same way, so that once the index has been created, the search algorithm can be run with either implementation without the necessity for recreating the index.

In theory, the transition matrix implementation should have better performance than the adjacency list for automaton simulation, because it has $\mathcal{O}(||a||)$ time for looking up $\delta(q_i, a)$ for any state q_i and label a. However, there are two possible issues with this implementation:

1. The asymptotic bound for the size of the transition matrix is $\mathcal{O}(|\Sigma| \times ||\mathit{pref}(()T)||)$. For practical examples, both the input alphabet and the length of the ranked prefix notation of the indexed tree T can be very large. At best, this can cause large number of cache misses when looking up the transitions, at worst it may prevent the index to be read into the memory. Some may point out that this problem would be eliminated by appropriate sparse implementation of the transition matrix. We argue,

Table 5.6: Linear index file layout

Type	Interpretation
`char[4] "LNDX"`	Index sigil for error checking and debugging purposes
`char[4] "NAME"`	Symbol table sigil
`uint32`	Symbol table size
`RankedSymbol[symTableSize]`	Symbol table items (see tbl. 5.3)
`char[4] "PSTR"`	Preorder string sigil
`uint32`	Preorder string length
`uint32[preorderLength]`	Preorder string itself
`char[4] "SJMP"`	Subtree transition table sigil
`uint32`	Subtree transition table size
`uint32[sTableSize]`	Subtree transition table items
`char[4] "AUTO"`	Suffix automaton sigil
`SuffixAutomaton`	Serialized suffix automaton (see tbl. **??**)
`char[4] "ORIG"`	Origins table sigil
`uint32`	Origins table size
`PString[]`	Origins table items (see tbl. 5.2)

Table 5.7: `SAState` – Suffix automaton state layout

Type	Interpretation
`uint32`	State's length label
`uint8`	Final state flag

Table 5.8: `SAEdge` – Suffix automaton edge layout

Type	Interpretation
`uint32`	Edge label offset
`uint32`	Edge label length
`uint32`	Index of the target state

however, that this would be essentially equivalent to a good adjacency list implementation.

2. Naive implementation of the transition table has $\mathcal{O}(|\Sigma|)$ time complexity for iteration over all outgoing edges of any state q_i. This can be easily remedied, however. The approach we've chosen exploits the unused elements of the transition matrix as links to the next *used*, or non-empty element of the matrix.

Table 5.9: Suffix automaton layout

Type	Interpretation
`uint32`	Number of states
`SAState[]`	State array
`char[4] "SGIL"`	State array sigil
	for each state:
`uint32`	Index of the source state
`uint32`	Number of edges
`SAEdge[]`	Edge array
`char[4] "SGIL"`	Edge array sigil

Indeed, experimental measurements have shown that in the final scheme, the adjacency list implementation of the automaton has better running times by approximately two orders of magnitude.

5.9 FLLI search function

The implementation of the search function is relatively straightforward, with the exception of the implementation of the *Rev* array. Like the bit vector masks in the WBC index, this array is likely to be very sparse and given the size of the datasets used for measurement in this work, it would be very inefficient to implement it as a plain array.

We have tested both hash tables and self-balancing search trees as implementations of the *Rev* and found no noticeable difference in performance between the two approaches, probably because the number of occurrences k is low enough that there is little difference between $\mathcal{O}(1)$ and $\mathcal{O}(\log k)$ lookup times.

However, either implementation was found about two orders of magnitude faster than the naive approach. Therefore, we have finally settled on hash table implementation.

CHAPTER 6

On position heaps

This chapter presents two algorithms and a new data structure that we found as a side product of this work. They were not placed in the *Basic notions and definitions* chapter, because, to our knowledge, they are not something that can be found in previous work. Neither could they be placed in any other chapter, because they haven't actually been used in this work in any way. Nevertheless, we considered them noteworthy enough to be included in this work as they are subject of future work and they fit into the department's research.

Consider again the example shown in figure 1.6 on page 16 and the similarities between the suffix tree and the position heap. Specifically, the set of all nodes and edges of the position heap are subsets of all nodes and edges of the corresponding suffix tree, with only labels of the nodes moved around. Labels of the edges remain the same (this is not the case for compact suffix trees). The algorithms 15 and 16 exploit this similarity and also the fact, that position heaps are, as their name subtly suggests, heaps. The algorithms simply "heapify" the suffix tree, moving the node labels upwards with the highest value labels being pulled to a parent node in case the parent has multiple labeled children. The remaining unlabeled nodes are then discarded.

Position heaps can be constructed by performing a lossy compaction on a suffix tree, as shown in algorithm 15. To construct position heap from a compact suffix tree, one could construct suffix tree from compact suffix tree, then run algorithm 15, or use more general algorithm 16 directly on the compact suffix tree. Both algorithms are given in a functional style in the sense they create new structure based on their input without modifying the original structure. However, it is possible to use them in-place with trivial modification.

Theorem 1. *Algorithm 15 is correct and will produce position heap of string x for any correct suffix tree of x.*

Algorithm 15 Construction of Position Heap From Suffix Tree

function SuffTreeToPosHeap(*node*)
  $r \leftarrow$ copy of *node*
  $e \leftarrow \{(l, \text{SuffTreeToPosHeap}(n)) \mid (l, n) \in \text{edges}(node)\}$
  if $\text{position}(r) = \text{nil}$ **then**
    $(l, m) \leftarrow \text{MaxNode}(e)$
    if $m = nil$ **then**
      return nil
    end if
    $\text{position}(r) \leftarrow \text{position}(m)$
    $\text{position}(m) \leftarrow \text{nil}$
    $\text{edges}(r) \leftarrow (e \setminus \{(l, m)\}) \cup \{(l, \text{SuffTreeToPosHeap}(m))\}$
  else
    $\text{edges}(r) \leftarrow e$
  end if
  return r
end function

function MaxNode(*edges*)
  if $edges = \emptyset \vee (\forall (label, node) \in edges)(\text{position}(node) = \text{nil})$ **then**
    return (nil, nil)
  end if
  return $(l, n)\,;\ (l, n) \in edges \wedge \text{position}(n) = \max\{\text{position}(m) \mid (k, m) \in edges\}$
end function

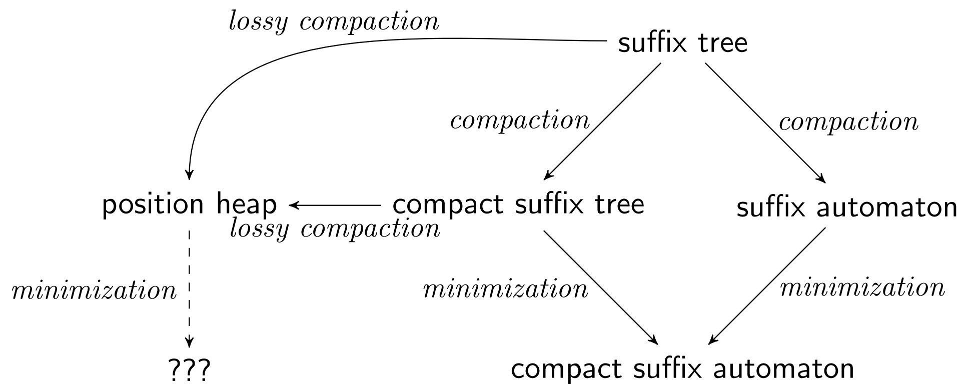


Figure 6.1: Conversions between suffix index structures

Algorithm 16 Construction of Position Heap From Compact Suffix Tree

```
function CSuffTreeToPosHeap(node)
    e ← ∅
    for all (label, child) ∈ edges(node) do
        child ← CSuffTreeToPosHeap(child)
        if length(label) > 1 then
            n ← new node with position position(child)
            position(child) ← nil
            l ← pref(label, 1)
            edges(n) ← {(l^{-1} label, child)}
            e ← e ∪ (l, CSuffTreeToPosHeap(n))
        else
            e ← e ∪ (label, child)
        end if
    end for

    r ← new node with position position(node)
    if position(r) = nil then
        (l, m) ← MaxNode(e)
        if m = nil then
            return nil
        end if
        position(r) ← position(m)
        position(m) ← nil
        edges(r) ← (e \ {(l, m)}) ∪ {(l, SuffTreeToPosHeap(m))}
    else
        edges(r) ← e
    end if
    return r
end function
```

Theorem 2. *Algorithm 16 is correct and will produce position heap of string x for any correct compact suffix tree of x.*

Proof of correctness of these algorithms, i.e. proof of theorems 1 and 2 is beyond the scope of this work, but is something we would like to address in the future.

It should be relatively simple to use these algorithms to derive algorithm for construction of position heaps from suffix automata, because the suffix automata are acyclic graphs and thus can be traversed in much the same way trees can (the only difference being that some nodes may be visited multiple times). The only problem is that the *position* function is not defined for states of suffix automaton as they are defined in most literature ([5]). Because some

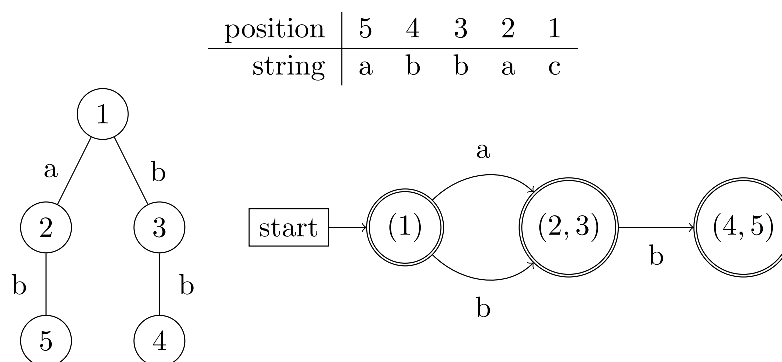


Figure 6.2: Example: position heap for the string "abbac" and an automaton produced by the heap's minimization

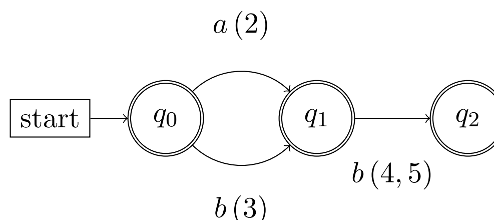


Figure 6.3: Modified version of position heap-derived automaton with position labels placed at the edges

states in suffix automaton for a string x correspond to multiple factors of x, they also correspond to multiple nodes of suffix tree for x and so function *position* would have to return a set rather than a scalar value.

Algorithms 15 and 16 are perhaps trivial, but to our knowledge, they are presented here for the first time. These algorithms change the situation shown in figure 1.5 to what is shown in 6.1. Given that by compaction and minimization of a suffix tree yields a compact suffix automaton, a question may be asked what is the result of heapifying suffix tree to get position heap and then minimizing that position heap. It is possible this could produce an automaton that could be used for approximate factor search in string x like a position heap, but would be the minimal equivalent of said position heap, having possibly less than $||x||$ states.

Theorem 3. *There exists a string x such that PH(x) can be minimized; that is minimization of PH(x) yields automaton $\mathcal{A}$ such that number of states of $\mathcal{A}$ is less than $||x||$.*

Figure 6.2 show example of a position heap for the string "abbac" and an automaton, which was created by the heap's minimization. Note that the

number of states of the automaton is 4, which is less than $||abbac|| = 5$. This proves theorem 3. In other words, position heaps which are not minimal exist.

Just like the position heap, the resulting automaton has no false negatives when searching for occurrences of a factor of the indexed string. On the other hand, it has more false positives. For example, for input string “ab”, the position heap in figure 6.2 yields set of occurrences $(2, 5)$ (position 1 is omitted, because there cannot be a factor of length 2 at position 1), while the automaton in the same figure yields $(2, 3, 4, 5)$. This can be partially remedied by moving the position labels from states to edges (this would also require a trivial modification of the search algorithm) as shown in figure 6.3.

This automaton would find occurrences $(2, 4, 5)$ for the same input string, still more than the the position heap, but fewer than the simple minimization of the position heap in figure 6.2.

Analysis and proofs of the properties of this data structure are matter of future work.

CHAPTER 7

Experimental results

Table 7.1 shows the XML data sets we have used for performance measurements. We have chosen them for their number of elements on the presumption that the larger the trees are, the more representative the results would be.

We have run tests on existing subtree search (search where the subtrees where actually known to be present in the tree), nonexistent subtree search and tree pattern search. For each test and dataset, we attempted to find 10 subtrees for each of the following size categories: 20 to 30, 50 to 60, 100 to 110, 200 to 220, 300 to 330, 400 to 450 and 500 to 600 nodes. We ran each of these queries 10 times for both the the WBC algorithm, FLLI algorithm and the position heap index algorithm and took arithmetic average of the times. The time measured includes simulation of the subtree pushdown automaton and the interpretation of the final state only.

The measurements were made on a computer with Intel Core i7-3820 3.60GHz 64-bit CPU and 18GB RAM.

Table 7.1: Datasets used for measurements

File	Description	Elements	Max. depth
`psd7003.xml`	Protein Sequence Database	21305818	7
`SwissProt.xml`	SwissProt database	2977031	5
`dblp.xml`	DBLP Computer Science Bibliography	3332130	6
`nasa.xml`	Astronomical data	476646	8
`treebank_e.xml`	Partially-encrypted treebank	2437666	36
`standard.xml`	Synthetic data generated by xmark	1666315	12

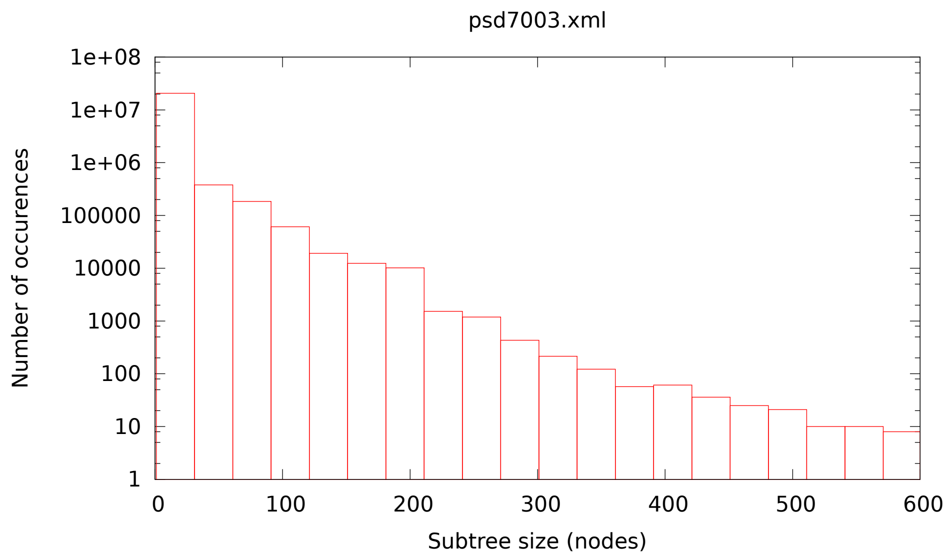


Figure 7.1: Subtree size frequency

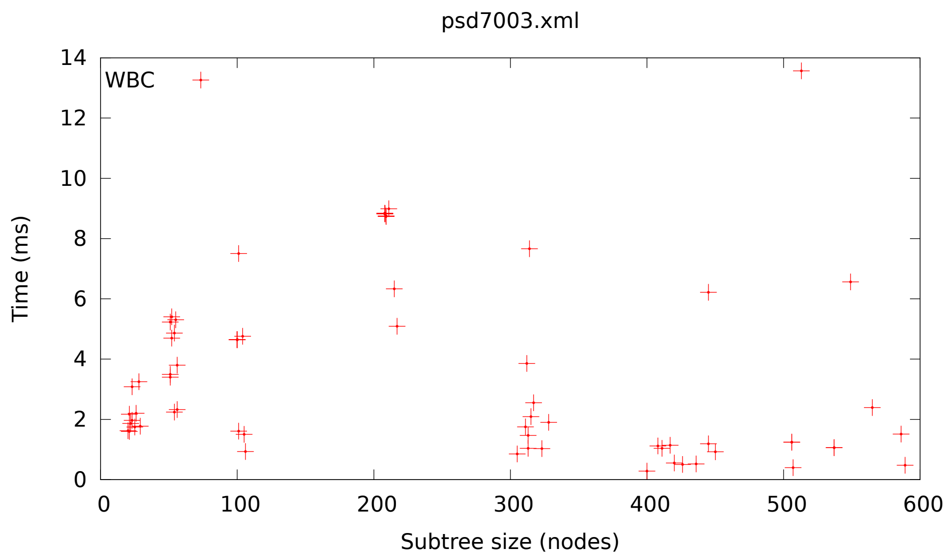


Figure 7.2: Existing tree search, WBC algorithm only

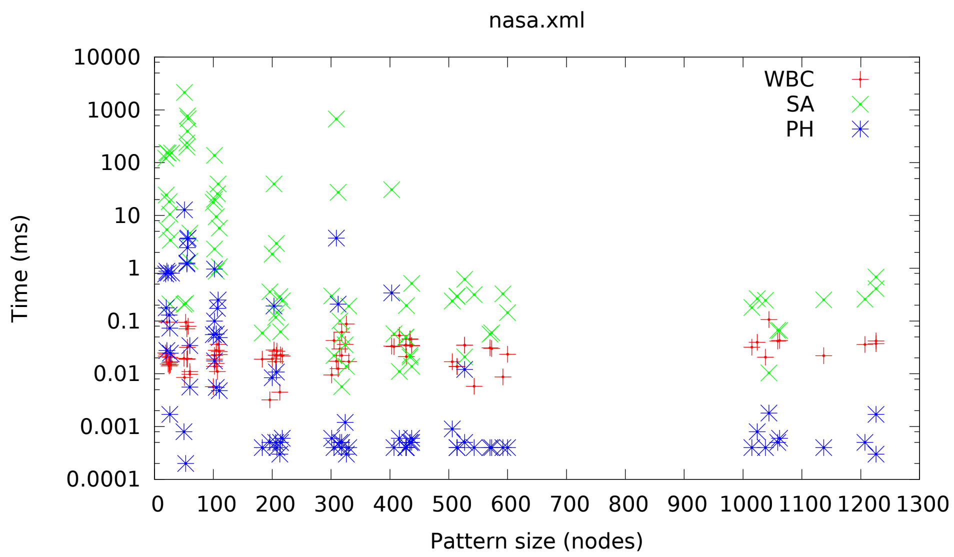


Figure 7.3: Subtree pattern search using WBC, FLLI and position heap

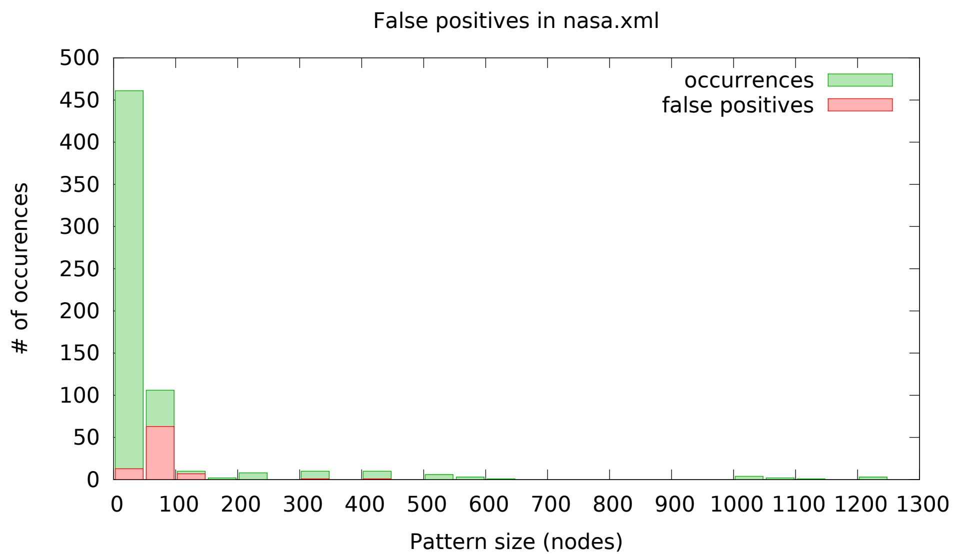


Figure 7.4: False positives and actual occurrences in position heap index

All algorithms exhibit non-linear relationship between the size of the searched subtree pattern and the measured time for the selected patterns and data. This is most likely because the time complexity of all of the measured algorithms is dependent on the number of occurrences and as the intuition suggests and figure 7.1 proves, there is an inverse relationship between the size of a subtree and the number occurrences of that subtree in a larger tree (at least in a real-world examples).

Figure 7.2 shows WBC results in more detail. There appears to be a local maximum around the subtree size of 200 nodes. Presumably, up to this size, the increasing size of the subtree had more influence on the running time than the decreasing number of subtree occurrences. For sizes larger than that, the running time decreased again until the number of occurrences reached a small integer close to 1 and couldn't decrease any further, at which point only the increasing subtree size could have any effect on the running time.

Figure 7.3 shows measured times of searching for the subtree patterns in the dataset `nasa.xml` for all algorithms. Interesting feature of the times to note is that just like with the frequency of the subtree sizes, there seems to be an inverse relationship between the size of the subtree pattern and the running time for both the FLLI and position heap index algorithms.

Figure 7.3 also shows that out of the three measured schemes, the position heap based index was found to be fastest for most inputs. Like the FLLI index it is more sensitive to the number of occurrences than the size of the pattern, so the WBC index is faster for some smaller patterns. The speed is at the cost of some false positives, but their number seems to be relatively low as shown in figure 7.4. The average false positive rate in the measured sample was ≈ 0.14 false positives per occurrence.

Conclusion

We have shown how an effective bit-parallel implementation of nondeterministic automata can, under certain circumstances, give better performance than comparable deterministic automata.

We have implemented three tree indexing schemes and used them index large real-world datasets. The WBC scheme, as it was originally described in [14], the FLLI scheme presented in [9] with only minor implementation modifications and finally we've implemented a new scheme, modification of the FLLI scheme which uses position heaps instead of compact suffix automatons.

We have experimentally compared the performance of their search phases and found that for most inputs, the new scheme employing position heaps has the best running times. However, the comparison is made more difficult by the fact that the running times of the algorithms react differently to varying subtree pattern size and number of occurrences of the pattern.

In the following paragraphs, we mention possible directions for future work.

Analysis of the presented position heap conversion algorithms

Algorithms 15 and 16 need proof of correctness and asymptotic bounds for their running time. If this time cannot be proven to be linear, it may be interesting to find a linear time algorithm that does the same.

New position heap-based data structure

The data structure created by the minimization of the position heap shown in figure 6.3 needs more analysis. We haven't given any asymptotic bounds for the number of states of this automaton, or the time of computing all occurrences of a pattern. The time is most likely going to be the same as for position heap, but a proof needs to be given.

Bibliography

[1] http://www.xml-benchmark.org/. 2011 [cit. 2012-05-10]. Dostupné z WWW: <`http://www.xml-benchmark.org/`>

[2] University of Washington XML Data Repository. 2012 [cit. 2012-05-07]. Dostupné z WWW: <`http://www.cs.washington.edu/research/xmldatasets/www/repository.html`>

[3] Bořivoj Melichar, T. P., Jan Holub: Text Searching Algorithms, Volume I: Forward String Matching. 2005.

[4] Crochemore, M.: Transducers and repetitions. *Theoretical Computer Science*, 1986: s. 63–86, ISSN 0304-3975, doi: http://dx.doi.org/10.1016/0304-3975(86)90041-1. Dostupné z WWW: <`http://www.sciencedirect.com/science/article/pii/0304397586900411`>

[5] Crochemore, M.; Vérin, R.: On Compact Directed Acyclic Word Graphs. In *Structures in Logic and Computer Science*, Springer-Verlag, 1997, s. 192–211.

[6] Domenico Cantone, E. G., Simone Faro: A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach.

[7] Ehrenfeucht, A.; McConnell, R. M.; Osheim, N.; aj.: Position heaps: A simple and dynamic text indexing data structure. *Journal of Discrete Algorithms*, 2011: s. 100 – 121, ISSN 1570-8667, doi:http://dx.doi.org/10.1016/j.jda.2010.12.001, 20th Anniversary Edition of the Annual Symposium on Combinatorial Pattern Matching (CPM 2009). Dostupné z WWW: <`http://www.sciencedirect.com/science/article/pii/S1570866710000535`>

[8] J. Janoušek, B. M.: Subtree Pushdown Automata and Tree Pattern Pushdown Automata for Trees in Prefix Notation. 2009.

[9] Janousek, J.; Melichar, B.; Polách, R.; aj.: A Full and Linear Index of a Tree for Tree Patterns. In *Descriptional Complexity of Formal Systems - 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, 2014,

s. 198–209, doi:10.1007/978-3-319-09704-6_18. Dostupné z WWW: <http://dx.doi.org/10.1007/978-3-319-09704-6_18>

[10] Janoušek, J.: Subtree Oracle Pushdown Automata for Ranked and Unranked Ordered Trees. 2009: s. 160–172.

[11] Kesheng Wu, A. S., Ekow J. Otoo: Optimizing bitmap indicies with efficient compression. *ACM Transactions on Database Systems (TODS)*, 2001, doi:10.1145/1132863.1132864. Dostupné z WWW: <http://dl.acm.org/citation.cfm?doid=1132863.1132864>

[12] Kesheng Wu, A. S., Ekow J. Otoo: A Performance Comparison of bitmap indexes. In *CIKM '01 Proceedings of the tenth international conference on Information and knowledge management*, 2001, doi:10.1145/502585.502689. Dostupné z WWW: <http://dl.acm.org/citation.cfm?doid=502585.502689>

[13] Melichar, B.: Arbology: Trees and Pushdown Automata. In *Language and Automata Theory and Applications: 4th international conference, LATA 2010, Trier, Germany*, Springer, 2010, s. 32–49.

[14] Milík, J.: *Simulation of a nondeterministic pushdown automaton for tree indexing*. Bachelor thesis, Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2012.

Index

$\mathcal{S}$-transition, 17
$\mathcal{S}$-symbol group, 31
Rev array, 29

adjacency list, 46
alphabet, 3

counter automaton, 7

FLLI, 29
formal language, 3
full and linear index, 29

Kleene closure, 3

position heap, 13
position heap index, 35
pushdown automaton, 6

subpattern, 30
subtree jump table, 29

transition mask, 21
transition matrix, 46

APPENDIX A

Contents of the enclosed CD