Preface

The 8th workshop **ITAT'08 – Information Technology – Applications and Theory** (`http://ics.upjs.sk/itat`) was held in Horský hotel Hrebienok, (`http://www.sorea.sk/Default.aspx?CatID=24&hid=4`) located 1280 meter above the sea level in High Tatras (`http://www.tatry.sk/`), Slovakia, in end of September 2008.

ITAT workshop is a place of meeting of people working in informatics from former Czechoslovakia (official languages for oral presentations are Czech, Slovak and Polish; proceedings papers are in English).

Emphasis is on exchange of information between participants, rather than make it highly selective. Workshop offers a first possibility for a student to make a public presentation and to discuss with the "elders". A big space is devoted to informal discussions (the place is traditionaly chosen at least 1000 meter above the sea level in a location not directly accessible by public transport).

Thematically workshop ranges from foundations of informatics, security, through data and semantic web to software engineering.
These proceedings consists of

- 8 original scientific papers

All papers were refereed by at least two independent referees. There were 41 submissions.

The workshop was organized by Institute of Informatics of University of P.J. Šafárik in Košice; Institute of Computer Science of Academy of Sciences of the Czech Republic, Prague; Faculty of Mathematics and Physics, School of Computer Science, Charles University, Prague and Slovak Society for Artificial Intelligence.

Partial support has to be acknowledged from projects of the Program Information Society of the Thematic Program II of the National Research Program of the Czech Republic 1ET100300419 "Intelligent models, algorithms, methods and tools for the semantic web realization" and 1ET100300517 "Methods for intelligent systems and their application in data mining and natural language processing".

Peter Vojtáš

Program Committee

Peter Vojtáš, (chair), *Charles University, Prague, CZ*
Gabriela Andrejková, *University of P.J. Šafárik, Košice, SK*
Mária Bieliková, *Slovak University of Technology in Bratislava, SK*
Leo Galamboš, *Charles University, Prague, CZ*
Ladislav Hluchý, *Slovak Academy of Sciences, Bratislava, SK*
Tomáš Horváth, *University of P.J. Šafárik, Košice, SK*
Karel Ježek, *The University of West Bohemia, Plzeň, CZ*
Jozef Jirásek, *University of P.J. Šafárik, Košice, SK*
Jana Kohoutková, *Masaryk University, Brno, CZ*
Stanislav Krajči, *University of P.J. Šafárik, Košice, SK*
Věra Kůrková, *Institute of Computer Science, AS CR, Prague, CZ*
Markéta Lopatková, *Charles University, Prague, CZ*
Ján Paralič, *Technical University in Košice, SK*
Dana Pardubská, *Comenius University, Bratislava, SK*
Martin Plátek, *Charles University, Prague, CZ*
Jaroslav Pokorný, *Charles University, Prague, CZ*
Karel Richta, *Czech Technical University, Prague, CZ*
Gabriel Semanišin, *University of P.J. Šafárik, Košice, SK*
Václav Snášel, *Technical University VŠB in Ostrava, CZ*
Vojtěch Svátek, *University of Economics in Prague, CZ*
Jiří Šíma, *Institute of Computer Science, AS CR, Prague, CZ*
Július Štuller, *Institute of Computer Science, AS CR, Prague, CZ*
Jakub Yaghob, *Charles University, Prague, CZ*
Filip Zavoral, *Charles University, Prague, CZ*
Stanislav Žák, *Institute of Computer Science, AS CR, Prague, CZ*
Filip Železný, *Czech Technical University, Prague, CZ*

Organizing Committee

Tomáš Horváth, (chair), *University of P.J. Šafárik, Košice, SK*
Hanka Bílková, *Institute of Computer Science, AS CR, Prague, CZ*
Peter Gurský, *University of P.J. Šafárik, Košice, SK*
Róbert Novotný, *University of P.J. Šafárik, Košice, SK*
Jana Pribolová, *University of P.J. Šafárik, Košice, SK*
Veronika Vaneková, *University of P.J. Šafárik, Košice, SK*

Organization

ITAT 2008 Information Technologies – Applications and Theory was organized by
University of P.J. Šafárik, Košice, SK
Institute of Computer Science, AS CR, Prague, CZ
Faculty of Mathematics and Physics, Charles University, Prague, CZ
Slovak Society for Artificial Intelligence, SK

Table of Contents

Extending Datalog to cover XQuery*

David Bednárek

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
david.bednarek@mff.cuni.cz

Abstract. *Datalog is a traditional platform in database research and, due to its ability to comprehend recursion, it seems to be a good choice for modeling XQuery. Unfortunately, XQuery functions have arguments carrying sequences; therefore, logic-based models of XQuery must be second-order languages and, consequently, Datalog is usually extended by node-set variables. In this paper, we suggest an alternative approach - extending Datalog by allowing structured variables in a form similar to Dewey numbers. This extension is then used to model the behavior of a XQuery program as a whole, using predicates that reflect the semantics of XQuery functions only in the context of the given program. This fact distinguishes our approach from traditional models that strive to comprehend the behavior of a function independently of its context. The advantage of this approach is that it uses the same means to model the structural recursion of documents and the functional recursion of programs, allowing various modes of bulk processing, loop reversal and other optimization techniques.*

1 Introduction

Contemporary XQuery processing and optimization techniques are usually focused on querying and, in most cases, ignore the existence of user-defined functions. In the era of XSLT 1.0, the implementation techniques had to recognize user-defined functions (templates) well (see for instance [3]); however, this branch of research appears discontinued as the community shifted to XQuery. The recent development in the area of query languages for XML shows that the XQuery language will likely be used as one of the main application development languages in the XML world [1]. In particular, intensive use of user-defined functions may be expected.

There were several attempts to apply Datalog or Datalog-like models to XPath or XQuery. There are also top-down approaches using structural recursion, i.e. strongly syntactically limited form of Horn clauses with function symbols [7]. More general forms, using first-order logic, were also used in the area of XML constraints [8].

In this paper, we (informally) define the language BTLog as an extension of Datalog. In the Section 3, an abstraction of an XQuery program as a forest is defined. In the fourth section, the principles of the transformation to BTLog is defined and shown on an example. In the Section 5, a detailed representation of the most important XQuery core operators is given.

* Project of the program "Information society" of the Thematic program II of the National research program of the Czech Republic, No. 1ET100300419

2 BTLog

Traditional Datalog program is a set of rules in the form of Horn clauses without function symbols. We will extend this definition with one binary function symbol T, corresponding to the creation of a binary tree $\mathsf{T}(x, y)$ from subtrees x and y. We will call this language *BTLog* (from *binary-tree* logic). Of course, addition of a function symbol raises the power of the language quite dramatically; therefore, some properties of Datalog are lost and new problems are raised:

- *Termination* – using the T-operator, any number of values may be generated. Therefore, termination is not generally guaranteed and any BTLog evaluation algorithm shall cope with termination problems.
- *Minimal model semantics* – without negation, minimal model semantics works well with BTLog, just as it works with Datalog without negation.
- *Stratification* – In Datalog$^\neg$, stratification is used to extend the notion of minimal model. Similar definition may be used in BTLog, resulting in the language BTLog$^{\neg,\mathrm{strat}}$.
- *Non-stratifiable program semantics – stable model* semantics is used for non-stratified BTLog$^\neg$ programs.

Definition of the abovementioned terms and detailed discussion of theoretical properties of such a language may be found for instance in [5].

3 Abstraction of a XQuery program

Similarly to the normative definition of the XQuery semantics, we use (abstract) grammar rules of the *core grammar* [9] as the base for the models. A XQuery program is formalized as a forest of abstract syntax trees (AST), one tree for each user-defined function and one for the main expression. Each node of each AST, i.e. each sub-expression appearing in the program, has a (program-wide) unique *address* E. These addresses will participate as subscripts in BTLog predicate names and they will also appear as constants in some rules.

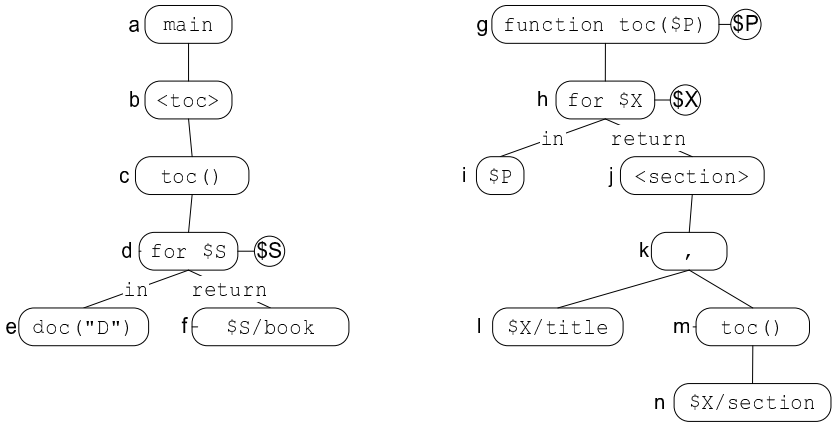

Fig. 2. Query 1 – Forest model.

```
declare function toc( $P)
{ for $X in $P return <section> {
   $X/title , toc( $X/section) } </section>
};

<toc> {
  toc( for $S in doc("D") return $S/book)
} </toc>
```

Fig. 1. Query 1.

Fig. 2 shows the abstract syntax forest corresponding to the Query 1 at Fig. 1. Node adresses are shown as letters left to the nodes.

For each AST node E, the set $\mathsf{vars}[E]$ contains the names of *accessible variables*. In particular, when E is the root of a function AST, $\mathsf{vars}[E]$ contains the names of arguments of the function, including implicit arguments like the context node.

4 Principles of the transformation

The model is based on the following principles:

- Nodes within a tree are identified by *node identifiers* using *Dewey ID* labeling scheme. (See, for instance [6].)
- A tree is encoded using a mapping of Dewey labels to *node properties*.
- A tree created during XQuery evaluation is identified by a *tree identifier* derived from the context in which the tree was constructed.
- A node is globally identified by the pair of a tree identifier and a node identifier.
- A sequence (i.e. any XQuery expression value) is modeled using a mapping of *sequence identifiers* to *sequence items*. Since a sequence may mix atomic values and document nodes, the mapping is divided into two interweaved *lists*.
- Each *sequence* containing document nodes is accompanied by a *tree environment* which contains the encoding of the document trees to which the nodes of the sequence belong.
- Evaluating a `for`-expression corresponds to iteration through all sequence identifiers in the value of the `in`-clause.
- A particular context reached during XQuery evaluation is identified by the pair of a *call stack*, containing positions in the program code, and a *control variable stack*, containing sequence identifiers selected by the `for`-expressions along the call stack.
- Node identifiers, tree identifiers, sequence identifiers, and control variable stacks share the same domain of binary trees with values on leaves, allowing to construct each kind of identifier from the others. In most cases, a binary tree is used to encode a (generalized) string – then, the rightmost path in the tree has the length of the string and the children of the rightmost path are the letters of the string.

4.1 Model predicates

Our model assigns a set of predicates to each AST node, i.e. to each address E:

- *Invocation* $\mathsf{inv}_E(i,f)$ enumerates the contexts in which the expression E is evaluated. Argument i represents the call stack that brought the execution to the examined expression E. f is the stack of sequence identifiers selected by the `for`-clauses throughout the descent along i to E. The two stacks together form the identification of the dynamic context in which an expression is evaluated. While the XQuery standard defines dynamic context as the set of variable assignments (with some negligible additions), our notion of

dynamic context is based on the stack pair that determines the descent through the code to the examined expression, combining both the code path stored in i and the `for`-control variables in f. The key to the sufficiency of this model is the observation that the variable assignment is a function of the stack pair.

- *Atomic list* $\mathsf{alst}_E(i, f, s, v)$ represents the atomic value portion of the assignment of the result value of the expression E to the contexts enumerate by $\mathsf{inv}_E(i, f)$. s is a sequence identifier, v is a value of an atomic type as defined by the XQuery standard. The predicate $\mathsf{alst}_E(i, f, s, v)$ is true if the value of the expression E in the context (i, f) contains the atomic value v at position s.
- *Node list* $\mathsf{nlst}_E(i, f, s, t, n)$ represents the node portion of the result value of the expression E. The meaning of i, f, and s is the same as in alst_E. t is a tree identifier – for external documents, it is a literal value, for temporal trees, it is the expression $\mathsf{T}(i_1, f_1)$ corresponding to the environment identification at the moment of tree creation. n is a node identifier in the form of a Dewey ID.
- *Environment* $\mathsf{env}_E(i, t, n, a)$ represents the tree environment associated to the result value of the expression E. i determines the call context (note that the environment is independent of the control variable stack). t is a tree identifier, n is a node identifier, a is a tuple of properties assigned to a node by the XML Data Model, containing node kind, name, typed and string values, etc. Particular properties are accessed using predicates $\mathsf{name}(a, v)$, $\mathsf{string}(a, v)$, etc.
- $\mathsf{valst}_{E,\$\mathtt{X}}(i, f, s, v)$, $\mathsf{vnlst}_{E,\$\mathtt{X}}(i, f, s, t, n)$, and $\mathsf{venv}_{E,\$\mathtt{X}}(i,t,n,a)$ represent the assignment of the values of the variable $\$\mathtt{X} \in \mathsf{vars}[E]$ to the contexts satifying $\mathsf{inv}_E(i, f)$. The meaning of the arguments is the same as in alst_E, nlst_E, and env_E.

4.2 Example

The following example shows the Query 1 transformed to a BTLog program. The subscripts in the predicate names correspond to the adresses shown in Fig. 2; unused and identity rules were removed. The main expression of the Query 1 transforms to the following BTLog rules:

$\mathsf{inv_a}(1, 1)$. -- program start

$\mathsf{env_e}(i, D, n, a) \text{ :– } \mathsf{inv_e}(i, f), \mathsf{doc}(\text{"}D\text{"}, n, a).$
-- doc("D") tree environment

$\mathsf{vnlst}_{\mathsf{f},\$\mathtt{S}}(i, \mathsf{T}(s, f), 1, t, n) \text{ :– } \mathsf{nlst_a}(i, f, s, t, n).$
-- variable $S

$\mathsf{nlst_f}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{vnlst}_{\mathsf{f},\$\mathtt{S}}(i, f, s, t, m),$
$\mathsf{env_e}(i, t, n, \mathsf{T}(\mathsf{element}, \mathsf{book})),$
$\mathsf{child}(m, n).$
-- $S/book node

$\mathsf{nlst_d}(i, f, \mathsf{T}(s, r), t, n) \text{ :– } \mathsf{nlst_f}(i, \mathsf{T}(s, f), r, t, n).$
-- the result of the for-expression

$\mathsf{vnlst}_{\mathsf{g},\$\mathtt{P}}(\mathsf{T}(\mathsf{c}, i), f, s, t, n) \text{ :– } \mathsf{nlst_d}(i, f, s, t, n).$
-- argument $P in `toc`

$\mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(\mathsf{T}(\mathsf{c}, i), t, n, a) \text{ :– } \mathsf{env_e}(i, t, n, a).$
-- environment of $P in `toc`

$\mathsf{nlst_c}(i, f, s, t, n) \text{ :– } \mathsf{nlst_g}(\mathsf{T}(\mathsf{c}, i), f, s, t, n).$
-- the return value of `toc`

$\mathsf{env_c}(i, t, n, a) \text{ :– } \mathsf{env_g}(\mathsf{T}(\mathsf{c}, i), t, n, a).$
-- the return value environment

$\mathsf{nlst_b}(i, f, 1, \mathsf{T}(i, f), 1) \text{ :– } \mathsf{inv_a}(i, f).$
-- the <toc> node

$\mathsf{env_b}(i, \mathsf{T}(i, f), 1, \mathsf{T}(\mathsf{element}, \mathsf{toc})) \text{ :– } \mathsf{inv_a}(i, f).$

$\mathsf{env_b}(i, \mathsf{T}(i, f), \mathsf{T}(s, p), a) \text{ :– } \mathsf{nlst_c}(i, f, s, t, m),$
$\mathsf{env_c}(i, t, n, a), \mathsf{cat}(n, m, p).$
-- the <toc> node environment

$\mathsf{out}(i, t, n, a) \text{ :– } \mathsf{env_b}(i, t, n, a).$
-- the output tree

The following rules correspond to the function `toc`:

$\mathsf{inv_j}(i, \mathsf{T}(s, f)) \text{ :– } \mathsf{vnlst}_{\mathsf{g},\$\mathtt{P}}(i, f, s, t, n).$
-- the invocation of the return clause

$\mathsf{vnlst}_{\mathsf{j},\$\mathtt{X}}(i, \mathsf{T}(s, f), 1, t, n) \text{ :– } \mathsf{vnlst}_{\mathsf{g},\$\mathtt{P}}(i, f, s, t, n).$
-- variable $X

$\mathsf{nlst_l}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{vnlst}_{\mathsf{j},\$\mathtt{X}}(i, f, s, t, m),$
$\mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(i, t, n, \mathsf{T}(\mathsf{element}, \mathsf{title})),$
$\mathsf{child}(m, n).$
-- $X/title expression

$\mathsf{nlst_n}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{vnlst}_{\mathsf{j},\$\mathtt{X}}(i, f, s, t, m),$
$\mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(i, t, n, \mathsf{T}(\mathsf{element}, \mathsf{section})),$
$\mathsf{child}(m, n).$
-- $X/section expression

$\mathsf{vnlst}_{\mathsf{g},\$\mathtt{P}}(\mathsf{T}(\mathsf{m}, i), f, s, t, n) \text{ :– } \mathsf{nlst_n}(i, f, s, t, n).$
-- argument $P in `toc`

$\mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(\mathsf{T}(\mathsf{m}, i), t, n, a) \text{ :– } \mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(i, t, n, a).$
-- environment of $P in `toc`

$\mathsf{nlst_m}(i, f, s, t, n) \text{ :– } \mathsf{nlst_g}(\mathsf{T}(\mathsf{m}, i), f, s, t, n).$
-- the return value of `toc`

$\mathsf{env_m}(i, t, n, a) \text{ :– } \mathsf{env_g}(\mathsf{T}(\mathsf{m}, i), t, n, a).$
-- the return value environment

$\mathsf{nlst_k}(i, f, \mathsf{T}(1, s), t, n) \text{ :– } \mathsf{nlst_l}(i, f, s, t, n).$

$\mathsf{nlst_k}(i, f, \mathsf{T}(2, s), t, n) \text{ :– } \mathsf{nlst_m}(i, f, s, t, n).$
-- the concatenated value

$\mathsf{env_k}(i, t, n, a) \text{ :– } \mathsf{venv}_{\mathsf{g},\$\mathtt{P}}(i, t, n, a).$

$\mathsf{env_k}(i, t, n, a) \text{ :– } \mathsf{env_m}(i, t, n, a).$

-- the environment of the concatenation

$\mathsf{nlst_j}(i,f,1,\mathsf{T}(i,f),1) \coloneq \mathsf{inv_j}(i,f).$

-- the <section> node

$\mathsf{env_j}(i,\mathsf{T}(i,f),1,\mathsf{T}(\mathsf{element},\mathsf{toc})) \coloneq \mathsf{inv_j}(i,f).$

$\mathsf{env_j}(i,\mathsf{T}(i,f),\mathsf{T}(s,p),a) \coloneq \mathsf{nlst_k}(i,f,s,t,m),$
$\mathsf{env_k}(i,t,n,a),\mathsf{cat}(n,m,p).$

-- the <section> node environment

$\mathsf{nlst_h}(i,f,\mathsf{T}(s,r),t,n) \coloneq \mathsf{nlst_j}(i,\mathsf{T}(s,f),r,t,n).$

-- the result of the for-expression

$\mathsf{nlst_g}(i,f,s,t,n) \coloneq \mathsf{nlst_h}(i,f,s,t,n).$

-- the result of the function

$\mathsf{env_g}(i,t,n,a) \coloneq \mathsf{env_j}(i,t,n,a).$

-- the result environment

Figure 3 show the dependence graph for the predicates of Query 1. There are three strongly connected components (shown in bold) – the first one carries the environment of argument $P (i.e. the input document) down through the recursion of the function `toc`. The second component corresponds to the recursive descent of the variable $P through the document. The third component collects the constructed nodes back, unwinding the call stack.

5 Representation of Core XQuery Operators

There are several variants of *core* subsets of XQuery, including the *core grammar* defined in the W3C standard [9], the LixQuery framework [4], and others [2]. Since the XSLT and XQuery are related languages and the translation from XSLT to XQuery is known (see [2]), the model may be applied also to XSLT.

Note: Most XQuery operators do not change the assignment of variable values; therefore, we will omit the propagation rules in the subsequent description. We will also omit the rules for alst_E and $\mathsf{valst}_{E,\$x}$ whenever they are similar to nlst_E and $\mathsf{vnlst}_{E,\$x}$.

Function call – E_0 = `f(` E_1 `)`

Assume that E_f is the root of the function and $x is the name of the formal argument. The rules implement pushing the call address E_0 onto the call stack and popping it back upon return.

$\mathsf{inv}_{E_f}(\mathsf{T}(E_0,i),f) \coloneq \mathsf{inv}_{E_0}(i,f).$
$\mathsf{vnlst}_{E_f,\$x}(\mathsf{T}(E_0,i),f,s,t,n) \coloneq \mathsf{nlst}_{E_1}(i,f,s,t,n).$
$\mathsf{venv}_{E_f,\$x}(\mathsf{T}(E_0,i),t,n,a) \coloneq \mathsf{env}_{E_1}(i,t,n,a).$
$\mathsf{nlst}_{E_0}(i,f,s,t,n) \coloneq \mathsf{nlst}_{E_f}(\mathsf{T}(E_0,i),f,s,t,n).$
$\mathsf{env}_{E_0}(i,t,n,a) \coloneq \mathsf{env}_{E_f}(\mathsf{T}(E_0,i),t,n,a).$

For Expression – E_0 = `for $y in` E_{in} `return` E_{ret}

The for-expression generates a new dynamic context for each member of the sequence E_{in}; in our model, it is represented by pushing the sequence identifier s onto the control stack f:

$$\mathsf{inv}_{E_{\mathsf{ret}}}(i,\mathsf{T}(s,f)) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i,f,s,t,m).$$

At the same time, the variable $y is added to the dynamic context, defined as follows:

$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i,\mathsf{T}(s,f),\mathsf{one},t,n) \coloneq$
$\mathsf{nlst}_{E_{\mathsf{in}}}(i,f,s,t,n).$
$\mathsf{venv}_{E_{\mathsf{ret}},\$y}(i,t,m,a) \coloneq \mathsf{env}_{E_{\mathsf{in}}}(i,t,m,a).$

For older variables, the following rules are defined for each $\$x \in \mathsf{vars}[E_0] \setminus \{\$y\}$:

$\mathsf{vnlst}_{E_{\mathsf{ret}},\$x}(i,\mathsf{T}(s,f),r,u,m) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i,f,s,t,n),$
$\mathsf{vnlst}_{E_0,\$x}(i,f,r,u,m).$
$\mathsf{venv}_{E_{\mathsf{ret}},\$x}(i,u,m,a) \coloneq \mathsf{venv}_{E_0,\$x}(i,u,m,a).$

Finally, the value of the for-expression is created by the concatenation of the return clause values:

$\mathsf{nlst}_{E_0}(i,f,\mathsf{T}(s,r),t,n) \coloneq \mathsf{nlst}_{E_{\mathsf{ret}}}(i,\mathsf{T}(s,f),r,t,n).$
$\mathsf{env}_{E_0}(i,t,n,a) \coloneq \mathsf{env}_{E_{\mathsf{ret}}}(i,t,n,a).$

Let Expression – E_0 = `let $y :=` E_{def} `return` E_{ret}

The let-expression adds the variable $y to the dynamic context. Nevertheless, the identification of the context is not changed and the other variables are also preserved.

$\mathsf{inv}_{E_{\mathsf{ret}}}(i,f) \coloneq \mathsf{inv}_{E_0}(i,f).$
$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i,f,s,t,n) \coloneq \mathsf{nlst}_{E_{\mathsf{def}}}(i,f,s,t,n).$
$\mathsf{venv}_{E_{\mathsf{ret}},\$y}(i,t,m,a) \coloneq \mathsf{env}_{E_{\mathsf{def}}}(i,t,m,a).$

Where Clause – E_0 = `for $y in` E_{in} `where` E_{wh} `return` E_{ret}

Adding where clause to a for-expression affects the set of contexts generated for the return clause; similarly, variable models are affected:

$\mathsf{inv}_{E_{\mathsf{ret}}}(i,\mathsf{T}(s,f)) \coloneq \mathsf{alst}_{E_{\mathsf{wh}}}(i,f,s,\mathsf{true}).$
$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i,\mathsf{T}(s,f),\mathsf{one},t,n) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i,f,s,t,n),$
$\mathsf{alst}_{E_{\mathsf{wh}}}(i,f,s,\mathsf{true}).$
$\mathsf{vnlst}_{E_{\mathsf{ret}},\$x}(i,\mathsf{T}(s,f),r,u,m) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i,f,s,t,n),$
$\mathsf{alst}_{E_{\mathsf{wh}}}(i,f,s,\mathsf{true}),\mathsf{vnlst}_{E_0,\$x}(i,f,r,u,m).$

Equality Test – $E_0 = E_1$ `eq` E_2

$\mathsf{eq}_{E_0}(i,f) \coloneq \mathsf{alst}_{E_1}(i,f,s,v),\mathsf{alst}_{E_2}(i,f,r,v).$
$\mathsf{alst}_{E_0}(i,f,1,\mathsf{true}) \coloneq \mathsf{eq}_{E_0}(i,f).$
$\mathsf{alst}_{E_0}(i,f,1,\mathsf{false}) \coloneq \neg\mathsf{eq}_{E_0}(i,f).$

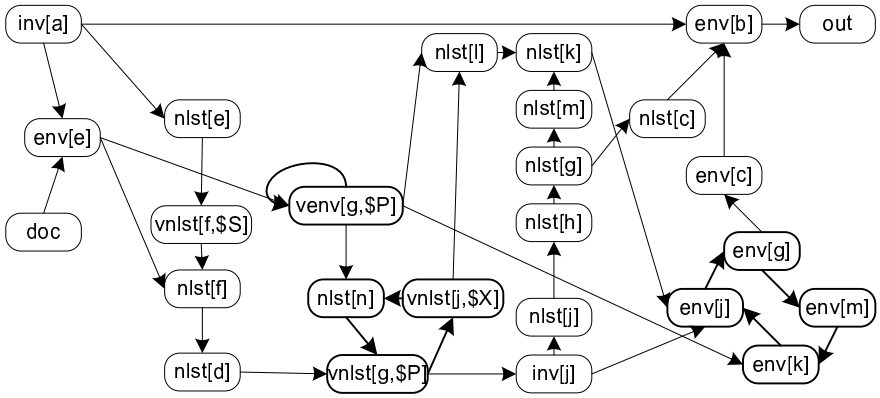

Fig. 3. Query 1 – Predicate dependence graph.

Node Construction – E_0 = <A>{ E_1 }</A>

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{one}, \mathsf{T}(i, f), \mathsf{one}) \text{ :– } \mathsf{inv}_{E_0}(i, f).$$
$$\mathsf{env}_{E_0}(i, \mathsf{T}(i, f), \mathsf{one}, a) \text{ :– } \mathsf{inv}_{E_0}(i, f), \mathsf{element}_{\mathtt{A}}(a).$$
$$\mathsf{env}_{E_0}(i, \mathsf{T}(i, f), \mathsf{T}(s, p), a) \text{ :– } \mathsf{nlst}_{E_1}(i, f, s, t, m), \mathsf{env}_{E_1}(i, t, n, a), \mathsf{cat}(n, m, p).$$

The auxiliary predicate cat corresponds to the concatenation of Dewey identifiers $n = m.p$ and it is defined as follows:

$$\mathsf{cat}(p, \mathsf{one}, p).$$
$$\mathsf{cat}(\mathsf{T}(s, n), \mathsf{T}(s, m), p) \text{ :– } \mathsf{cat}(n, m, p).$$

Navigation – E_0 = E_1 / *axis*::*

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{nlst}_{E_1}(i, f, s, t, m), \mathsf{env}_{E_1}(i, t, n, a), \mathit{axis}(m, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_1}(i, t, n, a).$$

The selection operator is driven by a predicate *axis* corresponding to the *axis* used in the navigation operator. These predicates are defined as follows:

$$\mathsf{child}(\mathsf{one}, \mathsf{T}(s, \mathsf{one})).$$
$$\mathsf{child}(\mathsf{T}(s, m), \mathsf{T}(s, n)) \text{ :– } \mathsf{child}(m, n).$$
$$\mathsf{parent}(m, n) \text{ :– } \mathsf{child}(n, m).$$
$$\mathsf{descendant}(\mathsf{one}, \mathsf{T}(s, n)).$$
$$\mathsf{descendant}(\mathsf{T}(s, m), \mathsf{T}(s, n)) \text{ :– } \mathsf{descendant}(m, n).$$
$$\mathsf{ancestor}(m, n) \text{ :– } \mathsf{descendant}(n, m).$$
$$\mathsf{descendantorself}(m, n) \text{ :– } \mathsf{cat}(n, m, p).$$
$$\mathsf{ancestororself}(m, n) \text{ :– } \mathsf{descendantorself}(n, m).$$

Node-Set Union – $E_0 = E_1$ `union` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{nlst}_{E_1}(i, f, s, t, n).$$
$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_1}(i, t, n, a).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_2}(i, t, n, a).$$

Note that the sequence identifiers s are not referenced at the head of the rule; instead, the identifier $\mathsf{T}(t, n)$ representing document order is used.

Node-Set Intersection – $E_0 = E_1$ `intersection` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{nlst}_{E_1}(i, f, r, t, n), \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_1}(i, t, n, a), \mathsf{env}_{E_2}(i, t, n, a).$$

Node-Set Difference – $E_0 = E_1$ `except` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \text{ :– } \mathsf{nlst}_{E_1}(i, f, r, t, n), \neg\mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_1}(i, t, n, a).$$

Concatenation – $E_0 = E_1$, E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(\mathsf{one}, s), t, n) \text{ :– } \mathsf{nlst}_{E_1}(i, f, s, t, n).$$
$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(\mathsf{two}, s), t, n) \text{ :– } \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_1}(i, t, n, a).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \text{ :– } \mathsf{env}_{E_2}(i, t, n, a).$$

Note that whenever the two environments env_{E_1} and env_{E_2} contain the same tree identifier t, the corresponding

tree information is merged by the env_{E_0} rules. Since the tree identifier exactly determines the context in which the tree was created, trees having the same identifier must be identical; therefore, merging the tree environments do not alter them anyway.

6 Conclusion

We have presented a model of XQuery evaluation based on Horn clauses under the BTLog$^\neg$ syntax. From the syntactic point of view, this model comprehends the following XQuery structures: Declare function, function call, for-clause, let-clause, where-clause, stable-order-by-clause, quantified expressions, equality operator on atomic values, Boolean operators including negation, `union`, `intersection`, `except` operators, concatenation (`,`) operator, statically named document references (`fn:doc`), forward/reverse axis navigation, name tests, `fn:root`, and element constructors.

There are two important omissions from the core XQuery that are not covered by this model: Positional variables and aggregate functions. There are also some flaws in error handling, namely the fact that the model may silently process some situations that shall be signalled as an error. These issues will be addressed by our future research.

The universal quantified expression (`every`), the equality operator, and the node-set subtraction operator (`except`) involve negation in their BTLog rules. Some XQuery programs are not stratifiable after conversion to BTLog$^\neg$. This is not necessarily a weakness of the approach – since the XQuery language is Turing-complete, we shall not expect general stratifiability. This way, the stratifiability of its BTLog$^\neg$ equivalent may be used as a borderline between "easy" and "difficult" cases. Fortunately, it shows that most of the real-life XQuery programs fall in the "easy" stratifiable category – for instance, all the XML Query Use Cases [10] programs are stratifiable.

Since termination in XQuery is not guaranteed, it may be expected that it is not generally guaranteed also in BTLog. Our future research will focus on static analysis methods trying to discover a termination guarantee in a BTLog program (of course, due to the Turing-completeness, no method can decide on the existence of a termination guarantee).

References

1. Chamberlin, D.: XQuery: Where Do We Go from Here? In: XIMEP 2006, 3rd International Workshop on XQuery Implementation, Experiences and Perspectives. ACM Digital Library, New York (2006)
2. Fokoue, A., Rose, K., Siméon, J., Villard, L.: Compiling XSLT 2.0 into XQuery 1.0. In: WWW '05: Proceedings of the 14th International Conference on World Wide Web, pp. 682–691, ACM, New York (2005)
3. Groppe, S., Böttcher, S., Birkenheuer, G., Höing, A.: Reformulating XPath Queries and XSLT Queries on XSLT Views. Technical report, University of Paderborn (2006)
4. Hidders, J., Michiels, P., Paredaens, J., Vercammen, R.: LixQuery: A Formal Foundation for XQuery Research. SIGMOD Rec., 34(4):21–26, ACM, New York (2005)
5. Hinrichs, T., Genesereth, M.: Herbrand Logic. Technical report LG-2006-02, Stanford University (2006)
6. Lu, J., Ling, T.W., Chan, C.-Y., Chen, T.: From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In: VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 193–204. ACM, New York (2005)
7. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion In: The VLDB Journal, pp. 76-110, Springer-Verlag (2000)
8. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-Variable Logic on Data Trees and XML Reasoning In: PODS'06, ACM, New York (2006)
9. XQuery 1.0 and XPath 2.0 Formal Semantics, W3C (2007)
10. XML Query Use Cases, W3C (2007), `http://www.w3.org/TR/xquery-use-cases/`

Determining XSLT streamability using new hierarchical XSD model*

Jana Dvořáková and Filip Zavoral

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic
{Jana.Dvorakova, Filip.Zavoral}@mff.cuni.cz

Abstract. *We introduce a new compact model of XML Schema called a schema tree. Given an XSLT transformation xsl and an XML schema xsd, we present a method which statically analyzes the schema tree constructed according to xsd and determines whether xsl can be processed in a streaming manner on a set of XML documents defined by xsd. We consider streaming processing that uses a stack of the size proportional to the depth of the input document - this processing is highly efficient in practice since real-world XML documents are shallow. The schema analysis is performed via stepwise application of templates of xsl on the schema tree. We present the implementation of a schema tree and the static XSLT analyzer on .NET platform.*

1 Introduction

Many applications need to employ streaming approach when processing huge data in XML format. Most typically, the languages XSLT [10] and XQuery [13] are used to specify XML transformations. Both of them enable the user to write a high-level specification based on tree manipulation. Common processors of these languages (e.g., Saxon, Xalan, AltovaXML) are tree-based, i.e., read the whole input document into memory and then perform the transformation itself.

The XSLT and XQuery tree-based processors are apparently not suitable when transforming XML streams or huge XML data. In this case, the transformation can be either written by hand using an event-base parser (e.g., SAX, StAX) or using some streaming transformation language (STX [1], XStream [5]). In both cases, writing the specification is a non-trivial task since the user must explicitly handle storing parts of the input document in the memory buffers for later processing.

In this paper we focus on the problem how to enable the user to write a tree manipulation specification in the XSLT language, and at the same time to process it in a streaming manner automatically. Such automatic streaming processor is supposed to apply the tree-manipulation functions over a continuous stream of data while the buffering is treated automatically. An important issue is to design the processor in such a way that the size of memory buffers is minimized for the given transformation and the input document.

We describe the implementation of the Xord framework which represents a prototype XSLT automatic streaming processor. The Xord framework is based on the formal framework for streaming XML transformations introduced in [3]. The framework is capable to process automatically a class of top-down XSLT transformations which captures a significant number of practically needed XML transformations. The processing is done using a stack of the size proportional to the depth of the input XML document - such processing is highly efficient in practice since real XML documents are shallow [9].

We focus especially on the schema-based analyzer which represents a powerful tool used within the Xord framework to determine the most efficient way of processing the given XSLT transformation. For a given XSLT stylesheet xsl and an XML schema xsd[1], it automatically analyzes the memory usage of the streaming processing of xsl on a set of documents defined by xsd.

The existing models for XML schemas (DOM, .NET XmlSchema) appeared inconvenient for the purpose of the streamability analysis, we therefore introduce the Xord Schema Model - a new compact model for schema representation. The model is abstract, and thus not bounded to a particular schema language. However, in the prototype implementation we employ W3C XSD notation [11,12] for XML schemas.

Related work. Several streaming processors for XSLT and XQuery have been implemented. However, their efficiency was demonstrated only by experiments on a small number of XML transformations and input XML documents. It is thus not known how much memory is consumed on clearly characterized transformation classes.

XML Streaming Machine (XSM) [8] processes a subset of XQuery on XML streams without attributes and recursive structures. It is based on a model called XML streaming transducer. The processor have been tested on XML documents of various sizes against a simple query. Using XSM the processing time grows linearly with the document size, while in the case of standard XQuery processors the time grows superlinearly. More complex queries have not been tested.

BEA/XQRL [4] is a streaming processor that implements full XQuery. The processor was compared with Xalan-J XSLT processor on the set of 25 transformations and another test was carried on XMark Benchmarks. BEA

* This work was supported in part by the National programme of research (Information society project 1ET100300419).

[1] We use the term *XML schema* for a general schema for XML documents.

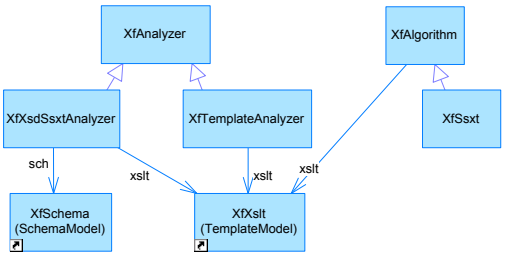

Fig. 1. The Xord framework.

processor was fast on small input documents, however, the processing of large documents was slower since the optimizations specially designed for XML streams are limited in this engine.

FluXQuery [7] is a streaming XQuery processor based on a new internal query language *FluX* which extends XQuery with constructs for streaming processing. XQuery query is converted into FluX and the memory size is optimized by examining the query as well as the input DTD. FluXQuery supports a subset of XQuery. The engine was benchmarked against XQuery processors Galax and AnonX on selected queries of the XMark benchmark. The results show that FluXQuery consumes less memory and runtime.

SPM (Streaming Processing Model) [6] is a one-pass streaming XSLT processor without an additional memory. Authors present a procedure that tries to converts a given XSLT stylesheet into SPM. No algorithm for testing the streamability of XSLT is introduced, and therefore the class of XSLT transformations captured by SPM is not clearly characterized.

2 Xord framework

The Xord framework for analyzing and transforming XML data is implemented on .NET platform. Its application interface is formed by a set of interface classes for traversing analyzed data structures. The core of the framework consists of these abstract models (see Fig. 1):

1. **Template Model** for transforming templates implemented by the *XfXslt* classes,
2. **Schema Model** for XML schemas implemented by the *XfSchema* classes,
3. **Algorithm Model** for streaming algorithms implemented by the *XfSsxt* classes,
4. **Analyzer Model** for static analyzers implemented by the *XfXsdSsxtAnalyzer* and *XfTemplateAnalyzer* classes.

Since the models are abstract, the Template Model may be adopted to model templates of any template-based XML transformation language and the Schema Model may be adopted to model any XML schema language based on structure definition.

Furthermore, the framework is complemented by a set of auxiliary helper classes. The algorithmic part of the API supports:

SsxtAlgorithm algorithm derived from the abstract Algorithm model, and

XsdSsxtAnalyzer algorithm derived from the abstract Analyzer model, and using the Schema Model and the Template Model.

The implementation of the above mentioned models are described in more detail in following sections.

3 XSLT representation

The Xord framework is currently restricted to process simple XSLT transformations on XML documents without data values.

Simple XSLT stylesheets. Simple XSLT stylesheet consists of an initializing template and several transforming templates. The initializing template sets the current mode to the initial mode m_0 and calls processing of the root element of the input document. It is of the form:

```
<xsl:template match="/">
    <xsl:apply-templates mode="m0"/>
</xsl:template>
```

The transforming templates are of the form:

```
<xsl:template match="name" mode="m">
     ... template body ...
</xsl:template>
```

The template body contains output elements (possibly nested) and apply-templates calls. Output elements are of the form:

```
<name>...element content...</name>
```

The `apply-templates` construct has a `select` attribute that contains selecting expression, and a `mode` attribute.

```
<xsl:apply-templates select="selexp" mode="m'"/>
```

A subset of XPath expression is allowed in templates - they contain child and descendant axis, and select nodes by name:

XPath := *Step* | *Step/XPath*
Step := `child::`*name* | `descendant::`*name*

where *name* refers to an element name.

Xord Template Model. In the Xord framework, XSLT stylesheets are represented by a set of classes, an *Xord Template Model*. Its simplified object structure is depicted in Fig. 2.

Each template from the XSLT contains a sequence of template calls. A template call consists of the parsed XPath expression and the template called by the *apply-templates* mechanism. The input template file is parsed into these structures before the analysis. Then the analysis algorithm directly traverses the DAG, evaluates the expressions etc.

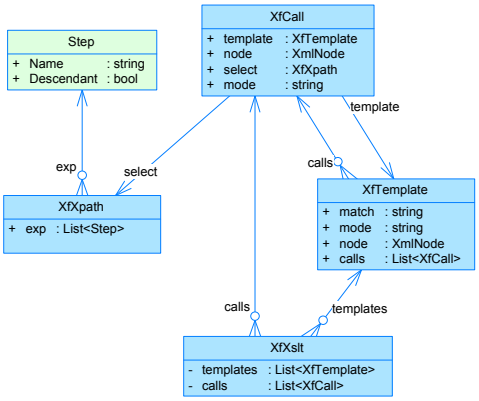

Fig. 2. The Xord Template Model.

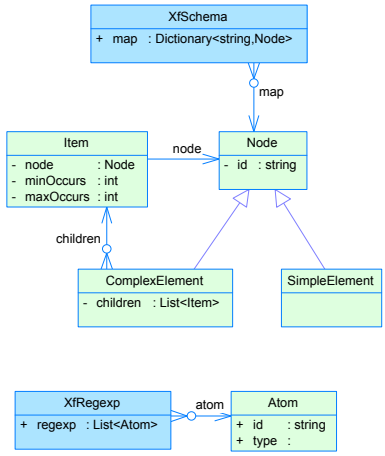

Fig. 3. The Xord Schema Model.

4 Hierarchical XML schema representation

We represent an XML schema hierarchically as a *schema tree*. The representation does not depend on a particular schema notation (DTD, XSD). The schema tree consists of two kinds of nodes:

- *element nodes*: correspond to an element type defined within schema
- *constructor nodes*: correspond to constructors used in the schema (sequence, choice, *, +, ?)

The relationships among element types and constructors are represented by the structure of the tree.

Some subtrees of schema tree may be identical - this situation occurs if we derive the schema tree from DTD or XSD containing shared element types. When designing the analyzer, the tree representation is more convenient. However in the implementation of schema-based analyzer each type is represented as a single node and the whole schema is represented as a DAG (see Schema Object Model below).

In the schema-based analysis, we consider XML schemas without the choice constructor and recursive definitions. Such schema can be represented as a single regular expression. This representation is useful in the extraction part of the analyzer algorithm (see Section 5).

Xord Schema Model. Although there are well established and widely used XML parsers, we have found no suitable parser for XSD. To perform schema manipulation, the .NET Framework provides a set of classes called the *Schema Object Model*, or SOM for short. The SOM is for schemas what DOM is for XML documents: the SOM classes represent various parts of a schema, for example *XmlSchemaSimpleType*, *XmlSchemaElement*, there are many other classes that represent attributes, facets, groups, complex types, and so on. This model is especially useful for creating schemas programmatically, but its application interface is not very useful for parsing and analyzing existing schemas.

Since the schema analysis using standard XML schema DOM model would be very complicated and tangled, we have designed an *Xord Schema Model* which is targeted to effective representation and analysis of existing schemas. A simplified object structure of that model is depicted in Fig. 3.

The whole schema is represented as an associative array of simple or complex type nodes. Each complex node contains a list of references to its child nodes with their cardinality. Using this recursive structure that form a DAG (or a tree with one particular node selected as a root), the parsed schema could be easily traversed and processed.

5 Schema-based analyzer

The schema-based analyzer applies the given XSLT stylesheet xsl to the schema tree xsd, starting at the root node. First, let us remind the principles of the XSLT application to the XML document tree. Let tmp be the current template of the XSLT stylesheet (at the beginning of the transformation, it is the template matching the root element in the initial mode m_0)

1. The node sequence selected by the XPath expressions in the rule calls of the current template are found.
2. The templates called by the rule calls are applied to the selected nodes.

However, in case of the schema tree, a modification of the first step of this simple algorithm is needed:

1. *All possible node sequences* selected by the XPath expressions in the rule calls of the current template are found.

2. The templates called by the rule calls are applied to the selected nodes.

Since the set of all possible node sequences selected by XPath expressions in the first step may be infinite, we represent it in the form of regular expression $regexp$. Such regular expression is basically a fragment of the schema tree, i.e., a set of its nodes (not necessarily connected) which is a fragment of the regular expression representing the whole schema tree.

The regular expression $regexp$ may contain both element nodes and constructor nodes. It is extracted as follows: First, the node sequence selected by the XPath expressions are found in the same way as in the XML document tree (constructor nodes are skipped). Second, all constructors appearing in the branch of the schema tree from the root to the selected nodes are added to the sequence. The hierarchy of the nodes is preserved by delimiting the nodes appearing at the same level of the schema tree by parentheses.

We say that $regexp$ represents possible *reading orders* of the element names selected by the expressions in tmp, i.e., the order in which the elements are accessed when a document defined by the schema xsd is read sequentially. Now let $names$ be a sequence of element names in the order they are called in tmp - clearly, the sequence can be constructed statically by examining the last steps of the XPath expressions in tmp. The $names$ sequence represents the *processing order* of the elements. In case one of the reading orders does not conform to the processing order, the order-preservation of the xsl is violated and the SSXT algorithm is not applicable[2]. It is thus only necessary to compare $regexp$ to the $names$ sequence in order to check applicability ot the stack-based algorithm.

Implementation. The core of the schema-based analyzer is the *AnalyzeNode* function which takes two arguments - a template of xsl and a node of the schema tree xsd. It performs the application of the template to the schema tree node as described above. Using the Template Model and the Schema Models allows the analyzer algorithm to be simple and straightforward - see Fig. 4.

The comparison of the $regexp$ to the $names$ sequence is accomplished by the $Compare$ function. Its implementation is based on inherent properties of its arguments. Instead of an expensive checking of swapping for each pair of names, the predicate is a compound of two simple steps. First, *regexp* is checked for existence of two distinct names within any '+' or '*' sequence. Second, the last names in $names$ are stripped to those contained in the schema being used, adjacent duplicities are reduced to a single name, and the resulting list is linearly compared to names contained in $regexp$. Since each name appearing $regexp$ must be contained in $names$, any difference cause a fail.

[2] See [3] for further details.

```
bool AnalyzeNode(XfTemplate t,XfSchema.Node n) {
    if(t.Empty)
        return true;
    XfLastNames li = t.GetLastNames();
    XfRegexp re = sch.ExtractFragment(n, t);
    if(re.Empty())
        return true;
    if(! sch.Compare(re, li))
        return false;
    foreach(XfCall call in t.calls) {
        List<XfSchema.Node> ln =
            new List<XfSchema.Node>();
        ln = sch.EvalExp(n, call.select);
        foreach(XfSchema.Node ni in ln) {
            AnalyzeNode(call.template, ni);
        }
    }
    return true;
}
```

Fig. 4. The code of the *AnalyzeNode* function.

6 Stack-based streaming algorithm

The stack-based algorithm is based on a formal model called *simple streaming XML transducer (SSXT)*, therefore we call it the *SSXT algorithm*. The transducer has a single input head that reads the input document sequentially, and a single output head that generates the output document sequentially. The SSXT is equipped with a stack to store temporary data.

The SSXT takes an input document d_{in} and a top-down XSLT stylesheet xsl as the input. It reads d_{in} sequentially in one pass and apply the stylesheet xsl stepwise. First, the template matching the root element of d_{in} in the initial mode m_0 is set to be the currently processed template (*current template*). The processing proceeds in cycles. During a single cycle, a single template call of the current template is processed.

Processing cycle. All XPath expression within a template are evaluating concurrently. The evaluation is realized by deterministic finite automata (DFA)[3]. A single DFA is constructed for each expression. When the processing of a template starts, the sequence of the initial states of DFAs is pushed on the stack. The input head of SSXT reads the elements of d_{in} in document order. When a start-tag is encountered, new sequence of DFAs is computed. Three situations may occur:

a) new sequence contains no final state - the input head continues in evaluation,
b) new sequence contains a single final state which belongs to the DFA evaluating the lastly-matched expression or an expression located *after* the lastly-matched expression - the corresponding template call is processed,

[3] We refer the reader to [2] for a more detailed description of this evaluating method.

c) new sequence contains a final state which belongs to the DFA evaluating expression located *before* the lastly-matched expression, or it contains two or more final states - error.

In case b), the current cycle configuration *(template id, matched expression id)* is pushed on the stack and new cycle for processing the called template starts. The cycle configuration is popped after the whole called template has been processed and the control moves back to the current template. In case a), the evaluation continues. Here if an end-tag is encountered, the sequence of the DFA states located at the top of the stack is popped. Hence, the XPath expression of the current template are evaluated on "branches" of d_{in}.

Implementation. The implementation of the SSXT algorithm uses both Template Model and Algorithm Model classes. Since the algorithm is stack-based, the main data structure used is a polymorphic stack stk of sequences of DFA states (*SIDfaSequence*) and cycle configurations (*SICycleConfig*), see Fig. 8.

Until the transformation is finished the top of stack is checked and the stack item is processed, see the function $RunSsxt$ in Fig. 5. In case of an empty stack and non-empty remaining input new DFA sequence is pushed on the stack.

```
void RunSsxt(XfXml xml)
{
  XfTemplate currTemplate = xslt.Start();
  XfCall currCall = null;
  bool transformed = false;
  while(!transformed) {
   if(!stk.Empty()) {
    switch(stk.Type()) {
     case XfStack.ItemType.DfaSequence:
      ProcessDfaSequence();
      break;
     case XfStack.ItemType.CycleConfig:
      ProcessCycleConfig();
      break;
    }
   } else {
    switch(xml.currType) {
     case XmlNodeType.Element:
      stk.Push(new SIDfaSequence(currTemplate));
      xml.Advance();
      break;
     case XmlNodeType.EndElement:
      currTemplate.Generate(currCall, null);
      transformed = true;
      break;
    }
   }
  }
}
```

Fig. 5. The code of the SSXT algorithm.

```
void ProcessDfaSequence(XfXml xml)
{
  SIDfaSequence ds = stk.GetDfaSequence();
  switch(xml.currType) {
   case XmlNodeType.Element:
    SIDfaSequence new_ds = ds.Transition(xml.currName);
    if(!new_ds.HasFinalStates()) {
     stk.Push(new_ds);
     xml.Advance();
    }
    else {
     XfCall myCall = new_ds.GetCallWithFinalState();
     currTemplate.Generate(currCall, myCall);
     XfTemplate calledTemplate =
       xslt.SelectTemplate(xml.currName, myCall.mode);
     if(calledTemplate.Empty) {
      calledTemplate.Generate(null, null);
      currCall = myCall;
      if(xml.laType == XmlNodeType.Element)
       stk.Push(new_ds);
      xml.Advance();
     } else {
      stk.Push(new SICycleConfig(currTemplate,
       myCall));
      currTemplate = calledTemplate;
      currCall = null;
     }
    }
    break;
   case XmlNodeType.EndElement:
    if(xml.laType == XmlNodeType.EndElement)
     stk.Pop();
    xml.Advance();
    break;
   default:
    stk.Pop();
    break;
  }
}
```

Fig. 6. The code of the *ProcessDfaSequence* function used in the SSXT algorithm.

The core of the DFA sequence processing (Fig. 6) is accomplished when start tags of elements are encountered. A new DFA sequence is generated on the stack in case the current DFA sequence contains no final states. Otherwise the output is generated and a new cycle configuration is placed on the stack. In case of a template without calls, its output is generated immediately.

The cycle configuration processing (Fig. 7) depends on the current XML node type. A start tag pushes a new DFA sequence while an end tag generates output.

7 Evaluation

The evaluation and measurements of the SSXT algorithm implementation confirmed our expectation that it requires a memory proportional to a depth of the input XML doc-

```
void ProcessCycleConfig(XfXml xml)
{
  SICycleConfig cc = stk.GetCycleConfig();
  switch(xml.currType) {
   case XmlNodeType.Element:
    if(xml.laType == XmlNodeType.Element)
     stk.Push(new SIDfaSequence(currTemplate));
    xml.Advance();
    break;
   case XmlNodeType.EndElement:
    currTemplate.Generate(currCall, null);
    currTemplate = cc.template;
    currCall = cc.call;
    stk.Pop();
    break;
  }
}
```

Fig. 7. The code of the *ProcessCycleConf* function used in the SSXT algorithm.

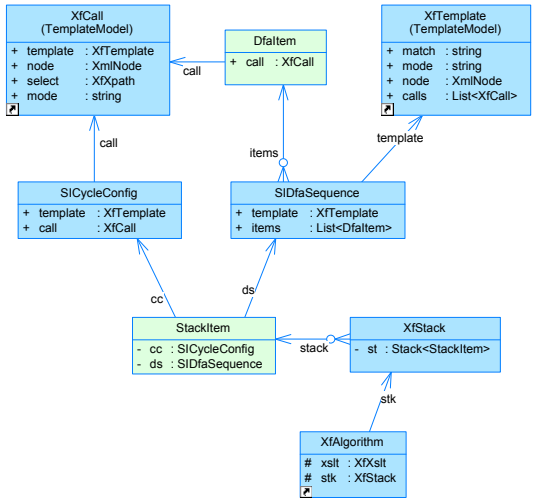

Fig. 8. The Xord SSXT Model.

ument. Since most documents are relatively shallow, our memory requirements are independent to the document size. Even for huge documents like DBLP, the SSXT algorithm required few hundreds KB while the commonly used XSLT processors like Saxon or Xalan crashed or hanged after allocating about 1.5 GB of memory.

8 Conclusion and future work

We have presented a prototype implementation of the Xord framework which represents an automatic streaming processor for the XSLT language. It incorporates a powerful schema-based analyzer which, for a given XSLT transformation xsl and an XML schema xsd, analyzes memory requirements of the streaming processing of xsl on a set of XML documents defined by xsd. The analyzer employs a special hierarchical model of XML schema called Xord Schema Model. We have implemented the Xord framework on .NET platform for a specific streaming processing using stack of the size proportional to the depth of the input XML document.

Our schema-based analyzer is restricted in several aspects - first, a subset of XSLT and XML schema definitions is considered, and second, it currently gives us only true/false answer whether the stack-based processing is applicable. However, we intend to extend it in the future research - if we examine particular pairs of elements for which the comparing function returns false and the possible size of their content, we may compute exact size of the memory buffers needed for processing such elements. Then it is only necessary to extend the basic stack-based streaming algorithm with such buffers and we obtain much more powerful automatic streaming XSLT processor.

References

1. O. Becker. Transforming XML on the Fly. In *Proceedings of XML Europe 2003*, 2003.
2. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
3. J. Dvořáková. Towards Analyzing Space Complexity of Streaming XML Transformations. In *The Second IEEE International Conference on Research Challenges in Information Science*. IEEE Computer Society, 2008.
4. D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL Streaming XQuery Processor. In *Proceedings of VLDB 2003*, pages 997–1008, 2003.
5. A. Frisch and K. Nakano. Streaming XML Transformations Using Term Rewriting. In *Proceedings of PLAN-X 2007*, 2007.
6. Z. Guo, M. Li, X. Wang, and A. Zhou. Scalable XSLT Evaluation. In *Advanced Web Technologies and Applications, LNCS 3007/2004*. Springer Berlin / Heidelberg, 2004.
7. C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An optimizing XQuery processor for streaming XML data. In *VLDB'2004: Proceedings of the Thirtieth International Conference on Very Large Databases*, pages 1309–1312, 2004.
8. B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of VLDB 2002*, pages 227–238, 2002.
9. I. Mlýnková, K. Toman, and J. Pokorný. Statistical Analysis of Real XML Data Collections. In *COMAD'06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing Company Limited.
10. W3C. *XSL Transformations (XSLT) Version 1.0, W3C Recommendation*, 1999. `http://www.w3.org/TR/xslt`.
11. W3C. *XML Schema Part 1: Structures Second Edition, W3C Recommendation*, 2004. `http://www.w3.org/TR/xmlschema-1`.
12. W3C. *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation*, 2004. `http://www.w3.org/TR/xmlschema-2`.
13. W3C. *XQuery 1.0: An XML Query Language, W3C Recommendation*, 2007. `http://www.w3.org/TR/xquery`.

Multiway blockwise in-place merging

Viliam Geffert and Jozef Gajdoš

Institute of Computer Science, P.J.Šafárik University, Faculty of Science
Jesenná 5, 041 54 Košice, Slovak Republic
viliam.geffert@upjs.sk, jozef.gajdos@upjs.sk

Abstract. *We present an algorithm for asymptotically efficient multiway blockwise in-place merging. Given an array A containing sorted subsequences $A_1, \ldots, A_k$ of respective lengths $n_1, \ldots, n_k$, where $\sum_{i=1}^{k} n_i = n$, we assume that extra $k \cdot s$ elements (so called buffer elements) are positioned at the very end of array A, and that the lengths $n_1, \ldots, n_k$ are positive integer multiples of some parameter s (i.e., multiples of a given block of length s). The number of input sequences k is a fixed constant parameter, not dependent on the lengths of input sequences. Then our algorithm merges the subsequences $A_1, \ldots, A_k$ into a single sorted sequence, performing $\Theta(\log k n) + O((n/s)^2) + O(s \log s)$ element comparisons and $3 \cdot n + O(s \cdot \log s)$ element moves.*[1]
Then, for $s = \lceil n^{2/3}/(\log n)^{1/3} \rceil$, this gives an algorithm performing $\Theta(\log k \cdot n) + O((n \cdot \log n)^{2/3})$ comparisons and $3 \cdot n + O((n \cdot \log n)^{2/3})$ moves. That is, our algorithm runs in linear time, with an asymptotically optimal number of comparisons and with the number of moves independent on the number of input sequences. Moreover, our algorithm is "almost in-place", it requires only k extra blocks of size $s = o(n)$.

1 Introduction

Given an array $A[1..n]$ consisting of sorted subsequences $A_1, \ldots, A_k$ each containing $n_1, \ldots, n_k$ elements respectively, where $\sum_{i=1}^{k} n_i = n$, the *classical multiway in-place merging* problem is to rearrange these elements to form a single sorted sequence of n elements, assuming that only one extra storage location (in addition to the array A) is available for storing elements. To store array indices, counters, etc. only $O(1)$ storage locations are available. The efficiency of a merging algorithm is given by two quantities: the number of pairwise element comparisons and the number of element moves carried out in the worst case, both expressed as a function of n. In merging, these are the only operations permitted for elements.

In this paper we study the computational complexity of the multiway *blockwise* in-place merging problem. More precisely, we assume that the entire array A is divided into blocks of equal size s, and that k extra blocks of size s are positioned at the very end of array A. Moreover, the lengths $n_1, \ldots, n_k$ of input sequences are positive integer multiples of s, and hence, there is always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$. We shall also assume, that before the merging starts, blocks can be mixed up quite arbitrarily, so we no longer know the original membership of blocks in the input sequences $A_1, \ldots, A_k$.

So far, the problem has been resolved for two-way merging, i.e., for $k = 2$ [4]. This algorithm uses $2n + o(n)$ comparisons, $3n + o(n)$ element moves and $O(1)$ extra locations for storing elements, in the worst case. Thus, by repeated application of this algorithm, we could carry out k-way merging in linear time, for arbitrary $k \geq 2$. However, implemented this way, the k-way merging would perform $3 \cdot \lceil \log k \rceil \cdot n + o(n)$ element moves and $2 \cdot \lceil \log k \rceil \cdot n + o(n)$ element comparisons. We shall show that the number of moves *does not depend on* k, if the lengths $n_1, \ldots, n_k$ are integer multiples of the block size s. Namely, using the algorithm of Geffert et. al [4] as our starting point, we show that multiway blockwise in-place merging is possible with $\lceil \log k \rceil \cdot n + O((n/s)^2) + O(s \cdot \log s)$ element comparisons and $3 \cdot n + O(s \cdot \log s)$ moves. For $s = \lceil n^{2/3}/(\log n)^{1/3} \rceil$, this gives an algorithm with $\lceil \log k \rceil \cdot n + O((n \cdot \log n)^{2/3})$ comparisons and $3 \cdot n + O((n \cdot \log n)^{2/3})$ moves, and the number of element moves independent on the number of input sequences. (It is also easy to show that the number of comparisons cannot be improved.)

2 Comparisons in a simple multiway merging

To explain how elements are compared, we first solve a simpler task. Assume that we are given an array A, consisting of k sorted subsequences $A_1, A_2, \ldots, A_k$, that are to be merged into a single sorted sequence. The lengths of these subsequences are $n_1, n_2, \ldots, n_k$ respectively, with $\sum_{i=1}^{k} n_i = n$.

Assume also that, together with the given array A, we are also given an extra array B of the same size n, which will be used as an output zone.

During the computation, the algorithm uses auxiliary index variables $i_1, \ldots, i_k$ and o_c, where i_j, for $j \in \{1, \ldots, k\}$, points to the smallest element of the sequence A_j not yet processed. This element will be called the *current input element of the j-th sequence*, or simply the *j-th input element*. The index o_c points to the leftmost empty position in the array B.

Then the straightforward implementation of the merge routine proceeds as follows. We find the smallest element not yet processed, by comparing elements at the positions

[1] Throughout the paper, $\log x$ denotes the binary logarithm of x.

$i_1, \ldots, i_k$, and move this element to the output zone in B. After that, we update the necessary index variables and repeat the process until all the elements have been merged. Implemented this way, each element will be moved just once and the number of comparisons, per each element, will be $k-1$. This gives us $(k-1)\cdot n$ comparisons and n element moves in total.

The number of comparisons can be reduced by implementing a selection tree of depth $\lceil \log k \rceil$ above the k current input elements. Initially, to build a selection tree, $k-1$ comparisons are required. Then the smallest element, not yet processed, can be moved to the output zone. After this, the element following the smallest element in the same subsequence is inserted in the tree and the selection tree is updated. To do this, only $\lceil \log k \rceil$ comparisons are needed. To avoid element moves, only pointers to elements are stored in the selection tree. (For more details concerning this data structure, see [1–3].) The number of moves remains unchanged, but now we have $k-1$ comparisons for the first element and only $\lceil \log k \rceil$ comparisons per each other element. This gives us a total of $(k-1) + \lceil \log k \rceil \cdot (n-1) \leq \lceil \log k \rceil \cdot n + O(1)$ comparisons.

3 Comparisons in a blockwise merging

This section describes one of the cardinal tricks used in our algorithm. Again, we are given the array A consisting of the sorted subsequences $A_1, \ldots, A_k$, to be merged together. We still have the extra array B, used as an output zone.

However, now the entire array A is divided into blocks of equal size s (the exact value of s will be determined later, so that the number of comparisons and moves is minimized) and, before the merging can start, these blocks are mixed up quite arbitrarily. Because of the permutation of blocks in A, we no longer know the original membership of blocks in the input sequences $A_1, \ldots, A_k$.

Still, the relative order of elements inside individual blocks is preserved. Moreover, we shall also assume that $n_1, \ldots, n_k$, the respective lengths of input sequences, are positive integer multiples of s, and hence, before mixing the blocks up, there was always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$.

Before passing further, we define the following relative order of blocks in the array A. Let X be a block with the leftmost and the rightmost elements denoted by x_{L} and x_{R}, respectively. Such block can be represented in the form $X = \langle x_{\text{L}}, x_{\text{R}} \rangle$. Similarly, let $Y = \langle y_{\text{L}}, y_{\text{R}} \rangle$ be an another block. We say that the block X is smaller than or equal to Y, if $x_{\text{L}} < y_{\text{L}}$, or $x_{\text{L}} = y_{\text{L}}$ and $x_{\text{R}} \leq y_{\text{R}}$. Otherwise, X is greater than Y. In other words, the blocks are ordered according to their leftmost elements and, in the case of equal leftmost elements, the elements at the rightmost positions are used as the second order criterion.

Now the modified merging algorithm proceeds as follows. First, using the above block ordering, find the smallest k blocks in the array A. These blocks will initially become the k *current input blocks*, their leftmost elements becoming the k *current input elements*. The j-th current input block will be denoted by X_j, similarly, the j-th current input element by x_j. The positions of current input elements are kept in index variables $i_1, \ldots, i_k$. Above the k current input elements, we build a selection tree. All blocks that are not input blocks are called *common blocks*.

After that, the merging process can proceed in the same way as described in Section 2. That is, using the selection tree, determine i_j, the position of the smallest input element not yet processed, among the k current input elements, and move this element to the output zone in the array B. Then the element positioned immediately on the right of x_j, within the same block X_j, becomes a new j-th current input element, its index pointer is inserted into the selection tree, and the tree is updated, with $\lceil \log k \rceil$ comparisons. This can be repeated until one of the current input blocks becomes empty.

When this happens, i.e., each time the element x_j, just moved to the output zone, was the last (rightmost) element in the corresponding input block X_j, the block X_j is "discarded" and the smallest (according to our relative block ordering) common block not yet processed will be used as the new j-th current input block. The leftmost element in this block will become the new j-th current input element. Since the blocks are mixed up in the array A, we need to scan sequentially all blocks (actually, all blocks not yet processed only) to determine which one of them is the smallest. This search for a new input block consumes $O((n/s)^2)$ additional comparisons: there are at most n/s blocks and such search is activated only if one of the input blocks has been discarded as empty, i.e., at most n/s times. (For the time being, just assume that we can distinguish discarded blocks from those not yet processed, at no extra cost.)

However, before merging, the blocks have been mixed up quite arbitrarily and hence their origin in the input subsequences $A_1, \ldots, A_k$ cannot be recovered. The proof that the above algorithm behaves correctly, that is, the elements are transported to the output zone in sorted order, will be published in the full version of the paper.

The number of element moves remains unchanged, but now we use $\lceil \log k \rceil \cdot n + O((n/s)^2)$ comparisons, under assumption that we can distinguish discarded blocks from those not yet processed at no extra cost.

4 In-place merging, simplified case

Now we shall convert the above merging algorithm into a procedure working "almost" in-place. More precisely, we are again given the array A containing the sorted subsequences $A_1, \ldots, A_k$, of respective lengths $n_1, \ldots, n_k$, with $\sum_{i=1}^{k} n_i = n$. All these lengths are positive integer multiples of the given parameter s.

However, we no longer have a separate array B of size n. Instead, we have some extra $k \cdot s$ elements positioned at the very end of the array A, behind A_k. The elements in this small additional area are greater than any of the elements in $A_1, \ldots, A_k$. During the computation, they can be mixed with other elements, but their original contents cannot be destroyed. These elements will be called *buffer elements*. To let the elements ever move, we have also one extra location where we can put a single element aside.

The sorted output should be formed within the same array A, in the locations occupied by the input sequences $A_1, \ldots, A_k$. (As a consequence, the buffer elements should also end up in their original locations.) Therefore, the moves are performed in a different way, based on the idea of internal buffering, used in a two-way in-place merging [4]. Nevertheless, the comparisons are performed in the same way as described in Section 3.

4.1 Initiation

Divide the entire array A into blocks of equal size s. Since the lengths of all input sequences $A_1, \ldots, A_k$ are positive integer multiples of s, there is always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$. Similarly, the buffer elements, positioned in the small additional area at the very end, form the last k blocks.

Initially, the last k blocks will be used as *free blocks*, their starting positions are stored in a *free block stack* of height k. After that, the position of one free block is picked out of the stack and this block is used as a so-called *escape block*. We also maintain a *current escape position* e_c, which is initially the position of the first (leftmost) element in the escape block. We create a hole here by putting the buffer element at this position aside.

Now, find the smallest k blocks[2] in the area occupied by $A_1, \ldots, A_k$, according to the relative block ordering defined in Section 3. This can be done with $O(k^2) \leq O(1)$ comparisons, by the use of some k cursors (index variables) moving along in A, since each of the sequences $A_1, \ldots, A_k$ is sorted. The smallest k blocks will initially become the k *current input blocks* $X_1, \ldots, X_k$. For each $j = 1, \ldots, k$, the first element x_j in the block X_j becomes a *j-th current input element*, and its position is kept in the index variable i_j. Above the k input elements, we build a selection tree of depth $\lceil \log k \rceil$. To do that, $k-1 \leq O(1)$ initial comparisons are needed.

The very first block of the array A becomes an *output block* and a position $o_c = 1$ pointing there becomes a *current output position*. The initial output position may — quite likely — coincide with a position of some current input element. Observe that $e_c \bmod s = o_c \bmod s$, which is an invariant we shall keep in the course of the entire computation. All other blocks are called *common blocks*.

In general, the algorithm maintains current positions of the following special blocks: free blocks, the number of which ranges between 0 and k, their leftmost positions are stored in the free block stack; exactly k input blocks, the current input positions inside the respective blocks are stored in the index variables $i_1, \ldots, i_k$; one output block with the current output position o_c inside this block; and one escape block with the current escape position e_c inside. The values of o_c and e_c are synchronized modulo s.

Usually, the optional free blocks, the k current input blocks, the output block, and the escape block are all disjoint, and the merging proceeds as described in Section 4.2. However, after the initiation, the output block may overlay one of the current input blocks, if the leftmost block in A_1 has been selected as an input block. If this happens, the current output position coincides with a position of one of the current input elements, and the computation starts in a very special mode of Section 4.9.

4.2 Standard situation

The standard situation is illustrated by Fig. 1. During the computation, the $k \cdot s$ buffer elements can be found at the following locations: to the left of the j-th input element x_j in the j-th input block X_j, for $j \in \{1, \ldots, k\}$, to the right of e_c in the escape block, with the hole at the position e_c, and also in free blocks, consisting of buffer elements only.

The elements merged already, from all the input blocks, form a contiguous output zone at the very beginning of A, ending at position $o_c - 1$. Hence, the next element to be output will go to the position o_c in the output block.

All elements not merged yet are scattered in blocks between the output zone and the end of the array A. The permutation of these blocks is allowed, however, elements to be merged keep their relative positions within each block. On the other hand, the origin of the blocks in the subsequences $A_1, \ldots, A_k$ cannot be recovered. So optional free blocks, input blocks, escape block, and common blocks can reside anywhere between the output zone and the end of the array A.

[2] Picking simply the leftmost blocks in the sequences $A_1, \ldots, A_k$ would do no harm. In addition, this would not require any initial element comparisons. However, we are presenting the algorithm in a form that is suitable for application in the general case, comparing elements in accordance with the strategy presented in Section 3.

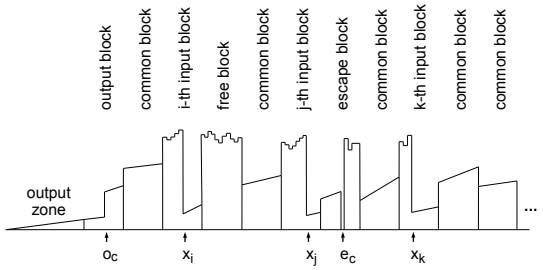

Fig. 1. Standard situation.

The output block spans across the current output position o_c, so its left part belongs to the output zone. As the output grows to the right, the elements lying to the right of o_c are moved from the output block to the corresponding positions in the escape block, i.e., to the right of e_c. The positions of o_c and e_c are synchronized, i.e., we have always $o_c \bmod s = e_c \bmod s$. Hence, the relative positions of escaping elements are preserved within the blocks. Moreover, o_c and e_c reach their respective block boundaries at the same time.

Now we are ready for merging. Using the selection tree, we determine x_j, the smallest element among the k current input elements in the blocks $X_1, \ldots, X_k$, and move this element to the output zone as follows:

Step A. The element at the position o_c in the output block escapes to the hole at the position e_c.
Step B. The smallest input element x_j not yet processed is moved from the position i_j to its final position at o_c.
Step C. A new hole is created at the position $e_c{+}1$ by moving its buffer element to the place released by the smallest input element just moved. After that, all necessary index variables are incremented and the selection tree is updated.

This gives 3 moves and $\lceil \log k \rceil$ comparisons per each element transported to its final location. Now there are various special cases that should be detected and handled with care. All exceptions are checked up on after the execution of Step B, in the order of their appearance, unless stated otherwise. Most of the exception handling routines replace Step C by a different action.

4.3 Escape block becomes full

If the rightmost element of the output block is moved to the last position of the escape block, the new hole cannot be created at the position $e_c{+}1$ in Step C. Instead, one free block at the top of the stack becomes the new escape block and a new hole is created at the beginning of this block. This is accomplished by removing its starting position from the free block stack and assigning it to e_c.

The subsequent move of the buffer element from the new position of e_c to the place released by the smallest input element does not increase the number of moves; it replaces the move in Step C. The selection tree is updated in the standard way.

It should be pointed out that, at this moment, there does exist at least one free block in the stack. Assume, for example, that the j-th input element x_j has just been transported to the output zone. After that, we have $r_j \in \{1, \ldots, s\}$ buffer elements in the j-th input block X_j, including the hole, but $r_h \in \{0, \ldots, s{-}1\}$ buffer elements in other input blocks X_h, for each $h \in \{1, \ldots, k\}$, $h \neq j$, since each input block, except for X_j, contains at least one input element. Moreover, the escape block is full, and hence it does not contain any buffer elements at all. Assuming there is no free block available, this gives at most $s{+}(k{-}1){\cdot}(s{-}1) < k{\cdot}s$ buffer elements in total. But this is a contradiction, since the number of buffer elements, including the hole, is always equal to $k{\cdot}s$.

4.4 Current input block becomes empty

We check next whether the smallest element x_j, just moved from the position i_j to the output zone, was the last element of the corresponding input block X_j. If so, we have an entire block consisting of buffer elements only, with hole at the end after Step B. This hole is filled in the standard way, described in Step C, but the old input block X_j becomes a free block and its starting position is saved in the stack. Since we have $k{\cdot}s$ buffer elements in total, a stack of height k is sufficient.

Next, we have to find a new j-th input block X_j, and assign a new value to i_j. Since the blocks are mixed up, we scan sequentially the remaining common blocks to determine which common block should become the new j-th current input block. The smallest common block, according to the block ordering introduced in Section 3, is the next one to be processed. As already shown in Section 3, the elements are transported to the output zone in sorted order even though this strategy does not necessarily pick up the j-th input block from the j-th input sequence A_j.

Free blocks, as well as all remaining current input blocks, are ignored in this scanning. Moreover, the elements to the left of e_c in the escape block (if not empty) together with the elements to the right of o_c in the output block are viewed as a single logical block. In a practical implementation, we can start with the leftmost escape-block element and the rightmost output-block element as a starting key and search the rest of the array for a common block with a smaller key.[3] If the logical block composed of the left part of the escape block and the right part of the output

[3] It is quite straightforward to detect whether a block beginning at a given position ℓ is common: the value of ℓ must not be saved in the free block stack, and $\lfloor \ell/s \rfloor$ must be different from

block should be processed next, the program control will be switched to the mode described in Section 4.5.

If the escape block is empty, then both e_c and o_c point to the beginning of their respective blocks. Then the escape block is skipped and the output block is handled as a common block, so we may even find out that the new input block should be located at the same position as the output block. This special mode is explained in Section 4.9.

The search for new input blocks costs $O((n/s)^2)$ additional comparisons: there are $O(n/s)$ blocks in total and such search is activated only if one of the input blocks is exhausted, i.e., at most $O(n/s)$ times. The same upper bound holds for arithmetic operations with indexes as well.

4.5 One of the input blocks overlays the escape block

If the common block that should be processed next is the logical block composed of the left part of the escape block and the right part of the output block, then both the new current input block X_j and the escape block are located within the same physical block. Here x_j is always positioned to the left of e_c and the buffer elements are both to the left of x_j and to the right of e_c.

Once the position of x_j is properly initiated, all actions are performed in the standard way described in Section 4.2. That is, the elements are transported from the output block to the position of e_c, from the input blocks to the position of o_c, and buffer elements from e_c+1 to locations released in the input blocks. Since e_c moves to the right "faster" than does i_j, this special case returns automatically to the standard mode as soon as e_c reaches a block boundary. Then the escape block separates from the current input block X_j as described in Section 4.3.

4.6 Output block overlays the escape block

Next we check whether the output zone, crossing a block boundary, does not bump into any "special" block. It is easy to see that this may happen only if e_c points to the beginning of the escape block that is empty, using the fact that the positions of o_c and e_c are synchronized and that the special handling of Section 4.3 is performed first.

Now consider that the output block overlays the escape block, i.e., they are both located within the same physical block. In this mode, we always have $o_c = e_c$. The element movement corresponds now to a more efficient scheme:

Step B'. The smallest input element x_j not yet processed is moved to the hole at the position $o_c = e_c$.

Step C'. A new hole is created at the position $o_c+1 = e_c+1$ by moving its buffer element to the place released by x_j. Then all necessary index variables are incremented and the selection tree is updated.

Step A is eliminated, since $o_c = e_c$. This mode is terminated as soon as o_c and e_c reach a block boundary. We also need a slightly modified version of the routine described in Section 4.4. If one of the input blocks becomes empty, it becomes free as usual, but the combined output/escape block is skipped in the search for the next input block.

4.7 Output block overlays a free block

If the output zone crosses a block boundary and the value of o_c is equal to some f_ℓ, the leftmost position of a block stored in the free block stack, the new output block and the corresponding free block are overlaid. This can be verified in $O(k) \le O(1)$ time. By the same argument as in Section 4.6, we have that e_c must point to the beginning of an empty escape block.

Therefore, we can easily swap the free block with the escape block by swapping the pointers stored in f_ℓ and e_c, since both these blocks contain buffer elements only. Second, one move suffices to transport the hole from one block to another. Note that this element move is for free, we actually save some moves because the next s transports to the output zone will require only $2s$ moves, instead of $3s$ as in the standard case. Thus, the program control is switched to the mode described in Section 4.6.

4.8 Output block overlays a current input block

If the output position o_c points to some X_j after crossing a block boundary, the output block overlays the j-th input block X_j. Again, by the argument presented in Section 4.6, o_c can point to the beginning of an input block only if e_c points to the beginning of an empty escape block. There are now two cases to consider.

First, if the j-th current input element x_j is the leftmost element of X_j, the program control is switched immediately to the special mode to be described in Section 4.9.

Second, if x_j is not the leftmost element of X_j, we dispose of the empty escape block as free by storing its starting position e_c in the stack, create a hole at o_c by moving a single buffer element from the position o_c to e_c, and overlay the output block by a new escape block, by assigning the value of o_c to e_c. The additional transportation of the hole is for free, not increasing the total number of moves, because we can charge it as (nonexistent) Step A for the next element that will be transported to the output zone. Since x_j is not placed at the beginning of the block, we can guarantee that at least one transport to the output will use only two moves in the next future.

$\lfloor i_1/s \rfloor, \ldots, \lfloor i_k/s \rfloor$ (excluding $\lfloor i_j/s \rfloor$), and also from $\lfloor e_c/s \rfloor$. For each given block, this can be verified in $O(k) \le O(1)$ time, performing auxiliary arithmetic operations with indexes only, but no element comparisons or moves.

This special mode can be viewed as if *three blocks* were overlaid, namely, the output, escape, and the current input block X_j. The buffer elements are between the hole at $e_c = o_c$ and the current input element x_j. The elements are moved according to Step B' and Step C' of Section 4.6. However, there is a different exception handling here.

(1) If the rightmost input element of this combined block has been transported to the output zone, then the input block X_j separates from the output/escape block, since we search for the next input block to be processed. But here, unlike in Section 4.4, the combined output/escape block is not disposed of as free, moreover, it is skipped out during the search. The program control is switched to the mode of Section 4.6 as the output and escape blocks are still overlaid.

(2) Let us now consider that this combined block becomes full. This may happen only if, for some $h \neq j$, an element x_h from another input block X_h is moved to the output zone and, after Step B', the output position o_c "bumps" into x_j. In this case, we take one free block from the top of the stack and change it into a new escape block. We definitely have at least one free block available, since we disposed one block as free at the very beginning of this mode. The hole, located in X_h at the position of the last element transported to the output, jumps to a position e_c in the new escape block, so that $e_c \bmod s = o_c \bmod s$. This move replaces Step C' for the last element just merged. Hence, it does not increase the total number of moves. Then we follow the instructions of Section 4.9.

4.9 Output zone bumps into a current input element

The program control can jump to this special mode from several different places (Sections 4.1, 4.4, and two different places in Section 4.8). In any case, we have an empty escape block, containing the hole and buffer elements only. The output block and a block X_j, which is one of the input blocks, are overlaid. Moreover, there is no room in between, the output position o_c is pointing to the current input element x_j. The position of hole in the escape block is synchronized with o_c, i.e., we have $e_c \bmod s = o_c \bmod s$.

As long as the elements to be output are selected in the input block X_j, they can be moved to the output zone. This needs no actual transportation, just the positions of o_c and i_j are moved synchronously to the right. To keep e_c synchronized with o_c, we move the hole along the escape block in parallel, which gives us one move per element. There are two ways out of this loop.

(1) If o_c and i_j reach the block boundary, we simply search for the next input block to be processed; the current configuration is the same as if, in the standard mode, o_c, e_c, and i_j reached the block boundaries at the same time (with the old input block X_j disposed of as free, by Section 4.4). Thus, unless something "exceptional" happens, the program control returns to the standard mode. (The possible exceptions are those discussed in Sections 4.6–4.8, and 4.10.) The single move required to place the hole back to the beginning of the escape block is for free, it substitutes Step C for the last element merged.

(2) If the element to be transported to the output zone is an element x_h from another input block X_h, for some $h \neq j$, some rearrangements are necessary. Recall that the hole position e_c in the escape block is synchronized with o_c, i.e., we have $e_c \bmod s = o_c \bmod s$. First, the input element x_j is moved from position o_c to position e_c. Now we can transport x_h to the output position o_c. Finally, a new hole is created[4] at the position e_c+1 by moving its buffer element to the place released by x_h.
The result is that the current input block X_j, overlaid by the output block, jumps and overlays the escape block. Thus, the control is switched to the mode of Section 4.5.
Clearly, this rearrangement needs only three moves. Since one more element has been transported to the output zone, the number of moves is the same as in the standard case.

4.10 Common blocks are exhausted

If one of the current input blocks becomes empty, but there is no common block to become a new input block, the above procedure is stopped. At this point, the output zone, consisting of the elements merged already in their final locations, is followed by a residual zone of size n' starting at the position o_c. This zone consists of the right part of the output block, $k-1$ unmerged input blocks, at most k free blocks, and one escape block. Thus, the total length of this residual zone is $n' \leq s + (k-1)\cdot s + k\cdot s + s = (2k+1)\cdot s$.

The residual zone can be sorted by the use of Heapsort (including also the buffer element put aside at the very beginning of the computation, to create a hole). This will cost only $O(k\cdot s\cdot\log(k\cdot s)) \leq O(s\cdot\log s)$ comparisons and the same number of moves [5–9]. Alternatively, we could also use an algorithm sorting in-place with $O(s\cdot\log s)$ comparisons but only $O(s)$ moves [10].

[4] Unless the position e_c+1 itself is across the block boundary. If x_j is moved to the rightmost position in the escape block, the escape block jumps immediately and one free block becomes a new escape block. This nested exception thus returns the algorithm to the standard mode; all "special" blocks now reside in pairwise disjoint regions. However, we jump to the point where the standard routine checks the exceptions of Sections 4.4–4.10. Among others, we have to check whether the input block X_h has not become empty, or if the output zone, just crossing a block boundary, has not bumped into any other "special" block again.

Now we are done: the buffer elements are greater than any other element, and hence the array now consists of the subsequences $A_1, \ldots, A_k$ merged into a single sorted sequence, followed by a sorted sequence of buffer elements.

4.11 Summary

Summing up the costs paid for maintaining the selection tree, transporting the elements to the output zone, searching for smallest input blocks, and for sorting the residual zone, it is easy to see that the above algorithm uses $\lceil \log k \rceil \cdot n + O((n/s)^2) + O(s \cdot \log s)$ element comparisons and $3 \cdot n + O(s \cdot \log s)$ moves. For $s = \lceil n^{2/3} / (\log n)^{1/3} \rceil$, this gives an algorithm with $\lceil \log k \rceil \cdot n + O((n \cdot \log n)^{2/3})$ comparisons and $3 \cdot n + O((n \cdot \log n)^{2/3})$ moves.

5 Conclusion

In this paper we have shown that k-way blockwise in-place merging can be accomplished efficiently with almost optimal number of element comparisons and moves. Moreover, the number of element moves is independent on k, the number of input sequences. Note that this algorithm does not merge stably, that is, the relative order of equal elements may not be preserved. Whether there exist a stable multiway blockwise in-place merging algorithm is left as an open problem.

We conjecture that, using the algorithm described here as a subroutine, it is possible to devise an asymptotically efficient multiway in-place merging algorithm. We dare to formulate this conjecture since the work on such algorithm is currently in progress.

References

1. Katajainen, J., Pasanen, T.: In-Place Sorting with Fewer Moves. Inform. Process. Lett. **70** (1999) 31–37
2. Katajainen, J., Pasanen, T., Teuhola, J.: Practical In-Place Mergesort. Nordic J. Comput. **3** (1996) 27–40
3. Katajainen, J., Träff, J. L.: A Meticulous Analysis of Mergesort Programs. Lect. Notes Comput. Sci. **1203** (1997) 217–28
4. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically Efficient In-Place Merging. Theoret. Comput. Sci. **237** (2000) 159–81
5. Carlsson, S.: A Note on Heapsort. Comput. J. **35** (1992) 410–11
6. Knuth, D. E.: The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Second edition (1998)
7. Schaffer, R., Sedgewick, R.: The Analysis of Heapsort. J. Algorithms **15** (1993) 76–100
8. Wegener, I.: Bottom-Up-Heapsort, a New Variant of Heapsort Beating, on an Average, Quicksort (If n Is Not Very Small). Theoret. Comput. Sci. **118** (1993) 81–98
9. Williams, J. W. J.: Heapsort (Algorithm 232). Comm. Assoc. Comput. Mach. **7** (1964) 347–48
10. Franceschini, G., Geffert, V.: An In-Place Sorting with $O(n \cdot \log n)$ Comparisons and $O(n)$ Moves. J. Assoc. Comput. Mach. **52** (2005) 515–37

Searching all approximate covers and their distance using finite automata*

Ondřej Guth, Bořivoj Melichar, and Miroslav Balík

Czech Technical University in Prague, Prague, Czech Republic
{gutho1,melichar,balikm}@fel.cvut.cz

Abstract. *Cover is a type of a regularity of strings. A restricted approximate cover w of string T is a factor of T such that every position of T lies within some approximate occurrence of w in T. In this paper, the problem of* all restricted smallest distance approximate covers of a string *is studied and a polynomial time and space algorithm for solving the problem is presented. It searches for all restricted approximate covers of a string with given limited approximation using Hamming distance and it computes the smallest distance for each found cover. The solution is based on a finite automata approach, that provides a straightforward way to design algorithms to many problems in stringology. Therefore it is shown that the set of problems solvable using finite automata includes the one studied in this paper.*

1 Introduction

Searching regularities of strings is used in a wide area of applications like molecular biology and computer–assisted music analysis. One of typical regularities is cover.

Finding exact covers is not sufficient in some applications, thus approximate covers have to be computed. In this paper, the Hamming distance is considered.

Exact covers were introduced in [1], an algorithm for computation of all exact covers in linear time was presented in [4]. An algorithm using finite automata approach to computation of all exact covers was introduced in [5].

The algorithm presented in [2] searches for one restricted smallest approximate cover (i.e. cover with the smallest distance), using dynamic programming. An algorithm using finite automata approach to computation all restricted approximate covers for Hamming, Levenshtein, and Damerau distance was introduced in [3].

This paper is organized as follows. In Section 2, some notations and definitions used in this paper are described. In Section 3, the algorithm for the problem is presented. In Section 4, the complexities of the algorithm are proven. In Section 5, experimental results are shown.

2 Preliminaries

An *alphabet* is a nonempty finite set of symbols, denoted by A. A *string* over an alphabet is a finite sequence of symbols of the alphabet. Empty string is an empty sequence of symbols, denoted by ε. An *effective alphabet* of a string T is a set of symbols that really occur in T. Only effective alphabet is considered in this paper. A *language* is a set of strings. A set of all strings over alphabet A is denoted by A^*. The length of a string w is denoted by $|w|$, the i–th symbol of w is denoted by $w[i]$. An operation *concatenation* is defined in this way: $x, y \in A^*$, concatenation of x and y is xy, may be denoted by $x.y$. An operation *superposition* is defined in this way: $x = pu, y = us$, superposition of x and y is pus. Suppose $u, w, x, T \in A^*$. w is a *prefix* of T if $T = wu$, w is a *suffix* of T if $T = uw$, and w is a *factor* (also called a substring) of T if $T = uwx$. A set of all prefixes of T is denoted by $Pref(T)$, a set of all suffixes of T is denoted by $Suff(T)$, and a set of all factors of T is denoted by $Fact(T)$.

A *deterministic finite automaton* (also called a deterministic finite state machine, denoted by DFA) is a quintuple (Q, A, δ, q_0, F), where Q is a nonempty finite set of states, A is an input alphabet, δ is a transition function, $\delta : Q \times A \mapsto Q$, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states.

A *nondeterministic finite automaton* without ε–transitions is a quintuple (Q, A, δ, q_0, F), where Q is a nonempty finite set of states, A is an input alphabet, δ is a transition function, where $\delta : Q \times A \mapsto \mathcal{P}(Q)$, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states. It is denoted by NFA.

A state q is a *successor* of state p of a deterministic finite automaton (Q, A, δ, q_0, F) if $q = \delta(p, a)$ for some $a \in A$. A state q_N is a successor of a state p_N of a NFA $(Q_N, A, \delta_N, q_{0N}, F_N)$ if $q \in \delta_N(p_N, a)$.

String $w = a_1 a_2 \dots a_{|w|}$ is said to be *accepted by a DFA* (Q, A, δ, q_0, F) if there exists a sequence $\delta(q_0, a_1) = q_1, \delta(q_1, a_2) = q_2, \dots, \delta(q_{|w|-1}, a_{|w|}) \in F$. String $w = a_1 a_2 \dots a_{|w|}$ is said to be *accepted by a NFA* (Q, A, δ, q_0, F) if there exists a sequence $\delta(q_0, a_1) = Q_1, \delta(q_1, a_2) = Q_2, \dots, \delta(q_{|w|-1}, a_{|w|}) \subseteq F$ for some $q_1 \in Q_1, \dots, q_{|w|-1} \in Q_{|w|-1}$. A language accepted by a finite automaton M is denoted by $L(M)$.

A *left language of a state* q of a nondeterministic finite automaton (Q, A, δ, q_0, F) is a set of strings $w = a_1 a_2 \dots a_{|w|}$, where for each w exists a sequence $\delta(q_0, a_1) = Q_1, \delta(q_1, a_2) = Q_2, \dots, \delta(q_{|w|-1}, a_{|w|}) = Q_{|w|}, q \in Q_{|w|}$ for some $q_1 \in Q_1, \dots, q_{|w|-1} \in Q_{|w|-1}$. A *left language of a state* q of a DFA (Q, A, δ, q_0, F) is a set of strings $w = a_1 a_2 \dots a_{|w|}$, where for

* This research was partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM 6840770014, by the Czech Science Foundation as project No. 201/06/1039, and by the Czech Technical University in Prague as project No. CTU0803113.

each w exists a sequence $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2, \ldots, \delta(q_{|w|-1}, a_{|w|}) = q$.

A *maxfactor of a state* q of a DFA (Q, A, δ, q_0, F) is the longest string of left language of q, denoted by $maxfactor(q)$. A *depth of a state* q of a DFA is the length of $maxfactor(q)$, denoted by $depth(q)$.

A DFA $M_D = (Q, A, \delta, q_0, F)$ is *equivalent* to a NFA $M_N = (Q_N, A, \delta_N, q_{0N}, F_N)$ if $L(M_N) = L(M_D)$. Subset construction may be used:

1. Set $Q = \{\{q_0\}\}$ will be defined, state $q_0 = \{q_{0N}\}$ will be treated as unmarked.
2. If each state in Q is marked then continue with step 4.
3. Unmarked state q will be chosen from Q and the following operations will be executed:
   (a) $\delta(q, a) = \bigcup \delta_N(p_N, a)$ for $p_N \in q$ and for all $a \in A$,
   (b) $Q = Q \cup \delta(q, a)$ for all $a \in A$,
   (c) state $q \in Q$ will be marked,
   (d) continue with step 2.
4. $F = \{q : q \in Q, p_N \cap F_N \neq \emptyset, p_N \in q\}$.

Using subset construction of M_D equivalent to M_N, every state $q_D \in Q$ corresponds to some subset of Q_N. This subset is called a *d–subset*, denoted by $d(q_D)$. Each element of the d–subset corresponds to some state of Q_N. Where no confusion arises, depth of a state corresponding to an element $r_j \in d(q_D)$ of d–subset $d(q_D)$ is simply denoted by r_j, as numeric representation of r_j corresponds to the depth. In the algorithms below, d–subset is supposed to be implemented as a list, preserving order of its elements. An element of the d–subset is denoted by r_i, where the subscript i means an index (order) of the element r_i within the d–subset.

A *distance* is the minimum number of editing operations that are necessary to convert a string x into a string y. The maximum allowed distance is denoted by k.

The *Hamming distance* between strings x and y is equal to the minimum number of editing operations replace that are necessary to convert x into y. The Hamming distance function is denoted by D_H.

String $w \in A^*$ is an *approximate prefix* of a string $T \in A^*$ with the maximum Hamming distance k if there exists string $p \in Pref(T)$ such that $D_H(w, p) \leq k$. String w is an *approximate suffix* of the string T if there exists string $s \in Suff(T)$ such that $D_H(w, s) \leq k$.

A *nondeterministic Hamming suffix automaton* M for a string T and distance k is such nondeterministic finite automaton without ε–transitions, that $L(M) = \{w : D_H(w, s) \leq k, s \in Suff(T)\}$. Such an automaton $M = (Q, A, \delta, q_0, F)$ may be constructed in this way:

1. Create a layer of $|T| + 1$ states:
   (a) each state q_i^0 corresponds to a position i in T (plus initial state q_0, thus $0 < i \leq |T|$),
   (b) for each state q_i^0 (but the last $q_{|T|}^0$) define transition $\delta(q_i^0, T[i]) = q_{i+1}^0$,
   (c) define the last state $q_{|T|}^0$ final (note that until now such automaton accepts exactly T).
2. Similarly, create a layer for each "number of errors" l, $1 \leq l \leq k$ (only exception: we do not need any state q_i^l for $l > i$).
3. For each state q_i^l (but the last $q_{|T|}$ in each layer and but the last layer) and for each symbol $a \in A, a \neq T[i]$ (not occurring in T at position i), define transition $\delta(q_i^l, T[i]) = q_{i+1}^{l+1}$.
4. Create "long" transitions from q_0: $\delta(q_0, a) = \{q_i^0 : a = T[i], a \leq i \leq |T|\} \cup \{q_i^1 : a \neq T[i], 1 \leq i \leq |T|\}$.

For example of a transition diagram of a nondeterministic Hamming suffix automaton see Fig. 1.

A *level* of a state of a nondeterministic Hamming suffix automaton corresponds to the number of errors, a *depth* of a state of this automaton is equal to the corresponding position in T.

Definition 1 (Restricted approximate cover). *Let T and w be strings. We say, that w is a* restricted approximate cover *of T with Hamming distance k if w is a factor of T and there exist strings $s_1, s_2, \ldots, s_r$ (all some substrings of T) such that:*

1. *$D_H(w, s_i) \leq k$ for all i where $1 \leq i \leq r$,*
2. *T can be constructed by superpositions and concatenations of copies of the strings $s_1, s_2, \ldots, s_r$.*

Note 1. An approximate cover is more general regularity than restricted approximate cover, because (unrestricted) approximate cover of T needs not be a factor of T. In this paper, only restricted approximate cover is considered.

Definition 2 (Restricted smallest distance approximate cover). *Let T and w be strings. We say, that w is a* restricted smallest distance approximate cover *of T with distance k if w is a restricted approximate cover of T with the distance k and there exists no $l < k$ such that w is a restricted approximate cover of T with the distance l.*

Problem 1 (All restricted smallest distance approximate covers of a string). Given string T over alphabet A, Hamming distance function D_H and distance k, find all restricted approximate covers of T and their smallest distances. A set of all restricted smallest distance approximate covers of string T under Hamming distance k is denoted by $covers_{H^k}(T)$.

As any approximate cover of a string T under Hamming distance is an approximate prefix and an approximate suffix of T (proven in [3]), an automaton accepting only such strings can be used.

Definition 3 (Approximate cover candidate automaton). *An* approximate cover candidate automaton *(Q, A, δ, q_0, F) for string $T \in A^*$, Hamming distance function D_H and the maximum distance k accepts set $W = \{w_1, w_2, \ldots, w_l\}$ of factors of T, where for each $w_i \in W$ holds:*

1. *there exists* $p \in Pref(T)$ *such that* $D_H(p, w_i) \leq k$, *and*
2. *there exists* $s \in Suff(T)$ *such that* $D_H(s, w_i) \leq k$.

In [3], a construction of an automaton accepting intersection of approximate prefixes and approximate suffixes is used for construction of a deterministic approximate cover candidate automaton. Although this is a straightforward idea, specialized method (more effective) is presented for Hamming distance in the following section.

3 Problem solution

The principle of the solution is following: first, we perform a subset construction of a deterministic cover candidate automaton from a nondeterministic Hamming suffix automaton for string T and k, as every $d(q)$ represents a set of positions of $w = maxfactor(q)$ within T. If we treat with $d(q)$ as with a sorted list (ordered by depths of its elements), each pair of subsequent elements represents positions of subsequent occurrences of w within T. When for such positions $i, j, i < j$ holds $j - i > |w|$, we know that w cannot be a cover of T. The distance of w is the minimum l such that it is possible to remove all elements $r \in d(q)$ having $level(r) > l$ and the previous condition holds.

In fact, it is not necessary to save complete deterministic automaton. Unlike in [3], we do not make construction of the deterministic cover candidate automaton and subsequent computation of covering. A depth–first search algorithm is used to perform subset construction and computation of covering and of the distance of each cover: in Algorithm 2, for each state and symbol, a successor q is generated, it is determined whether it represents a cover and the distance is computed. When q represents an approximate prefix, its successors are recursively generated and processed. Note that the set of final states of the deterministic approximate cover candidate automaton is not needed (it would contain all states having d–subsets containing element corresponding to some final state of the nondeterministic Hamming suffix automaton).

Distance l of each cover $w = maxfactor(q)$ may vary between 0 and k. Moreover, it cannot be less than level of the first or the last element of $d(q)$, because each cover must be an approximate prefix and suffix. Of course, it cannot be more than the maximum level of elements of $d(q)$. The Algorithm 1 removes all the elements having the maximum level but the first and the last element of $d(q)$, and tries whether w covers T without those removed positions.

Example 1. Let us have a string $T = aabccccb$ over alphabet $A = \{a, b, c\}$ and let us compute a set of all restricted smallest distance approximate covers of T under Hamming distance $k = 2$ using Algorithm 3.

Because of the distance 2, we are interested in covers of length at least 3 or having distance less than 2. We construct a nondeterministic Hamming suffix automaton M_S (see Fig. 1), then an approximate cover candidate automaton M is analysed (see Fig. 2).

Looking at the d–subset $\{3, 4'', 8''\}$, it represents an approximate prefix and suffix aab of length 3, but for its positions holds $8 - 4 \not\leq 3$, thus the factor aab is not an approximate cover of T with Hamming distance 2. Looking at the other d–subset $\{3'', 5'', 6', 7', 8\}$, it represents factor ccb, that covers T with Hamming distance 2. It is checked

Algorithm 1 Smallest distance of a cover of T.

Input: d–subset $d(q)$ representing a cover w of T.
Output: The smallest distance l of w.

$l_{\min} \leftarrow \max\{level(r_1), level(r_{|d(q)|})\}$
$l_{\max} \leftarrow \max_{r \in d(q)}\{level(r)\}$
$l \leftarrow l_{\max}$
repeat
  for all $r \in d(q) \setminus \{r_1, r_{|d(q)|}\} : level(r) = l$ **do**
    remove r from $d(q)$
  end for
  $l \leftarrow l - 1$
until $l \geq l_{\min}$ and for all $i = 2, 3, \ldots, |d(q)| : r_i - r_{i-1} \leq depth(q)$
$l \leftarrow l + 1$.

Algorithm 2 Process state of a deterministic approximate cover candidate automaton $M = (Q, A, \delta, q_0, F)$ constructed for string T and the maximum distance k from a nondeterministic Hamming suffix automaton $M_S = (Q_S, A, \delta_S, q_{0S}, F_S)$.

Input: State q_i having depth i and the d–subset $d(q_i)$.
Output: The temporary set of restricted smallest distance approximate covers c.

$c \leftarrow \emptyset$
for all $a \in A$ **do**
  create new state q, define $depth(q) = depth(q_i) + 1$
  for all r_s in $d(q_i)$ (in order as stored in $d(q_i)$) **do**
    append all $r_i \in \delta_S(r_s, a)$ to $d(q)$ in ascending order by $depth(r_i)$
  end for
  if for the first $r_1 \in d(q)$ holds $r_1 \leq depth(q_i)$ **then**
    if exists $r \in d(q)$ where $level(r) = 0$ within M_S **then**
      define $w = maxfactor(q) = maxfactor(q_i).a$
      if $r_{|d(q)|} \in F_S$ **then**
        if for all $i = 2, 3, \ldots, |d(q)| : depth(r_i) - depth(r_{i-1}) \leq depth(q)$ **then**
          define l the smallest distance of w (Alg. 1)
          if $|w| > k$ or $l < |w|$ **then**
            $c \leftarrow c \cup (w, l)$
          end if
        end if
      end if
      process state q (this algorithm), c' is result
      $c \leftarrow c \cup c'$
    end if
  end if
end for

Algorithm 3 Computation of a set of all restricted smallest distance approximate covers for string T and the Hamming distance k.

Input: String $T = a_1 a_2 \dots a_n$, the Hamming distance k.
Output: Set of all restricted smallest distance approximate covers $covers_{H^k}(T)$ of string T using the Hamming distance function D_H and the distance k.

1: $covers_{H^k}(T) \leftarrow \{(T, 0)\}$.
2: Construct nondeterministic Hamming suffix automaton $M_S = (Q_S, A, \delta_S, q_{0S}, F_S)$ for T and k.
3: Create state q_0 of the deterministic approximate cover candidate automaton $M(T) = (Q, A, \delta, q_0, F)$.
4: Define $maxfactor(q_0) = \varepsilon$.
5: Process state q_0 using Algorithm 2.
6: $covers_{H^k}(T)$ is the resulting set from the previous step.

whether it covers T with distance 1 (Alg. 1). As the first element of the d–subset has level equal to 2, $l_{\min}$ is equal to 2. The resulting set of the covers is $covers_{aabccccb^H}(2) = \{(ccb, 2), (aabccccb, 0)\}$.

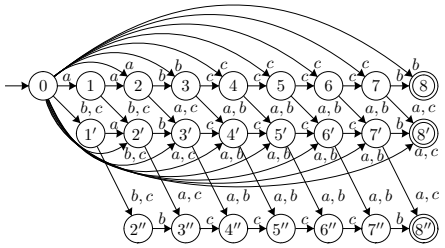

Fig. 1. Transition diagram of nondeterministic Hamming suffix automaton for string $aabccccb$ and the distance 2.

4 Complexities

Lemma 1. *The nondeterministic Hamming suffix automaton $M_S = (Q, A, \delta, q_0, F)$ for string T and the distance k contains $(|T| + 1) \cdot (k + 1) - \frac{k^2+k}{2}$ states and $|A| \cdot (|T| \cdot (k + 1) - 1 + \frac{k-k^2}{2}) + |T| - k + 1$ transitions.*

Proof. The automaton consists of layers of states $q^{(i)}$ for each level i. The layer of states q^0 contains $|T| + 1$ states. Each layer of states $q^{(i)}$ contains one state less in comparison with layer of states $q^{(i-1)}$, thus it contains $|T| - i + 1$ and layer of states $q^{(k)}$ contains $|T| - k + 1$ states.

The automaton contains $|A|$ transitions from each state, with some exceptions. There are $k + 1$ final states having no successor. In the layer of states $q^{(k)}$, each state has only one successor. From the initial state, there are $|T|$ transitions defined to the states $q^{(0)}$ having $level(q^{(0)}) = 0$ and $|T| \cdot (|A| - 1)$ transitions to the states $q^{(1)}$ having $level(q^{(1)}) = 1$. Thus in M_S there are $|Q|\cdot|A|+|T|\cdot|A|-(k+1)\cdot|A|-(|T|-k+1)\cdot(|A|-1) = |A|\cdot(|Q|-2)+|T|-k+1$ transitions.

Note 2. As restricted approximate covers of string T are exact factors of T, it is meaningful to consider effective alphabet A only, thus $|A| \leq |T|$ always holds. It is also meaningless to consider large k, because every factor of T having length less or equal to k is always approximate cover of T. Thus $k \leq |T|$ always holds.

Usually, $k \ll |T|$ and $|A| \ll |T|$ (e.g. in DNA analysis, $A = \{a, c, g, t\}$). Therefore k and $|A|$ may be considered as small constants independent of $|T|$.

Lemma 2. *The deterministic approximate cover candidate automaton M for string T and the Hamming distance contains at most $\frac{|T|^2+|T|}{2} + 1$ states.*

Proof. Each d–subset $d(q)$ of M contains at least one r such that $level(r) = 0$, thus $maxfactor(q) \in Fact(T)$. The number of possible factors of length $depth(q)$ is at most $|T| - depth(q) + 1$, thus the maximum number of states of M having equal depth is also $|T| - depth(q) + 1$. The automaton M also contains an initial state. Therefore, the number of states of M is at most $\frac{(|T|-1+1)+(|T|-|T|+1)}{2} \cdot |T| + 1$.

Lemma 3. *During the construction of the deterministic cover candidate automaton M for string T, Algorithms 2, 3 need to hold at most $|T| + 2$ states at a time.*

Proof. Algorithm 2 works as a depth–first search algorithm. For each state and symbol it generates at most one state – possible successor. Thus it holds at most $|T| + 1$ states of M ($|T|$ states having d–subsets representing exact prefixes of T plus initial state) and a state generated for a final state, having empty d–subset.

Lemma 4. *During the construction of the deterministic cover candidate automaton M for string T, Algorithms 2, 3 need to hold at most $\frac{|T|^2+|T|}{2} + 1$ elements of d–subsets at a time.*

Proof. Alg. 2 needs at most $|T| + 2$ states in a memory at a time (Lemma 3). The deterministic cover candidate automaton $M = (Q, A, \delta, q_0, F)$ is constructed by subset construction from a nondeterministic Hamming suffix automaton $M_S = (Q_S, A, \delta_S, q_{0S}, F_S)$. In M_S, each state but q_{0S} has at most one successor for each symbol, q_{0S} has $|T|$ successors for each symbol. For each state p_S and its successor q_S in M_S holds: $depth(q_S) > depth(p_S)$. The longest possible d–subset $d(p)$ contains $r_{|T|}$ having $depth(r_{|T|}) = |T|$, and r_1 having $depth(r_1) = 1$. As $|\delta_S(r_1, a)| \leq 1$ and $\delta_S(r_{|T|}, a) = \emptyset$ for every $a \in A$, for state p and its successor q in M holds: $|d(q)| \leq |d(p)|$ for $p \neq q_0$ and $|d(q)| \leq |T|$ for $p = q_0$.

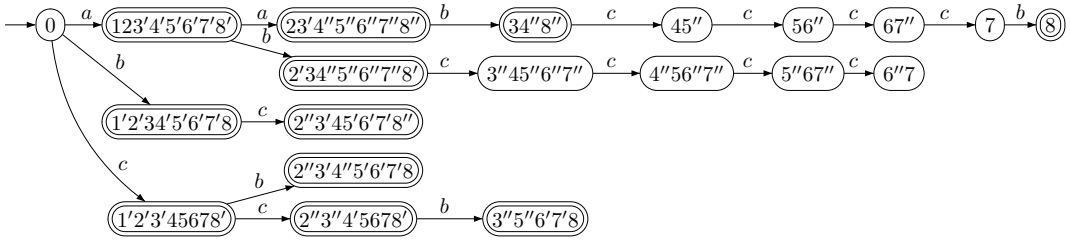

Fig. 2. Transition diagram of complete deterministic approximate cover candidate automaton for string $T = aabccccb$ and the maximum Hamming distance 2.

Theorem 1. *Space complexity of Alg. 3 is* $\mathcal{O}(|T|^2)$.

Proof. It clearly holds that for construction of the nondeterministic Hamming suffix automaton $M_S = (Q_S, A, \delta_S, q_{0S}, F_S)$, there is no need for any additional data structures. For the purpose of the construction of the deterministic cover candidate automaton M, only the set of states and transitions from q_{0S} need to be preserved, because the rest may be computed later in $\mathcal{O}(1)$ time and space using knowledge of a depth and a level of a state, k, and T. Thus the space complexity of this construction is $\mathcal{O}((k + |A|) \cdot |T|)$.

During the computation of the smallest distance (Algorithm 1), only $\mathcal{O}(1)$ additional data is needed. During the processing of states of M (Algorithm 2), the needed space is limited by the number of elements of all d–subsets (Lemma 4) preserved in a memory and by the number of all approximate covers (the result, limited by the number of all factors of T – at most $\mathcal{O}(|T|^2)$).

Lemma 5. *Using Algorithms 2, and 3 for construction of a deterministic cover candidate automaton* $M = (Q, A, \delta, q_0, F)$ *from a nondeterministic Hamming suffix automaton* $M_S = (Q_S, A, \delta_S, q_{0S}, F_S)$*, all d–subsets are sorted in ascending order by depths within* M_S.

Proof. Having $p, q \in Q \setminus \{q_0\}$ such that q is a successor of p, suppose that $d(p)$ is sorted in order by depths within M_S. It holds that for any $p_S, q_S \in Q_S$ such that q_S is a successor of p_S, $depth(q_S) > depth(p_S)$. Therefore $d(q)$ constructed from already sorted $d(p)$ is also sorted.

For $p = q_0$, it is supposed that $\delta_S(q_{0S}, a)$ is constructed as sorted in order by depths within M_S.

Lemma 6. *Time complexity of Algorithm 1 is* $\mathcal{O}(k \cdot |T|)$ *for each state.*

Proof. Algorithm 1 may remove some elements of a d–subset in each iteration, thus the iteration may take $\mathcal{O}(|T|)$ time. The number of iterations may be at most k.

Lemma 7. *Time complexity of Algorithm 2 (from the initial state) is* $\mathcal{O}((k + |A|) \cdot |T|^3)$.

Proof. Algorithm 2 constructs for all states q and all $a \in A$ the d–subsets of all possible successors of q. The number of states is $\mathcal{O}(|T|^2)$ (Lemma 2) and the number of elements of each d–subset is $\mathcal{O}(|T|)$. For each state, the computation of covering is performed (it takes $\mathcal{O}(|T|)$), and for each cover (their number is $\mathcal{O}(T^2)$), the computation of the smallest distance is performed (it takes $\mathcal{O}(k \cdot |T|)$ for each cover – Lemma 6).

Theorem 2. *Time complexity of Alg. 3 is* $\mathcal{O}((k+|A|)|T|^3)$.

Proof. It clearly holds that construction of the nondeterministic Hamming suffix automaton takes $\mathcal{O}((k+|A|)|T|)$. Construction of the deterministic cover candidate automaton takes $\mathcal{O}((k + |A|) \cdot |T|^3)$ (Lemma 7).

5 Experimental results

The algorithm was implemented in C++ using STL, the program was compiled using the GNU C++ compiler with O3 optimizations level. The dataset used to test the algorithm is the nucleotide sequence of Saccharomyces cerevisiae chromosome IV[1]. The string T consists of the first $|T|$ characters of the chromosome.

The first set of tests was run on a AMD Athlon 64 3200+ (2200 MHz) system, with 2.5 GB of RAM, under Fedora Linux operating system (see Figs. 3, 4).

The second set of tests was run on a AMD Athlon (1400 MHz) system, with 1.2 GB of RAM, under Gentoo Linux operating system (see Figs. 5, 6).

Note 3. In comparison with experimental results presented in [2], the algorithm presented in this paper runs a bit faster for the same data, even on a slightly slower computer (1.3 seconds in [2] for text length 100 vs. maximum 1.0 second for text length 114 – see Fig. 6).

6 Conclusion and future work

In this paper, we have shown that an algorithm design based on a determinisation of a suffix automaton is appropriate for all restricted smallest distance approximate

[1] The Saccharomyces cerevisiae chromosome IV dataset could be downloaded from http://www.genome.jp/.

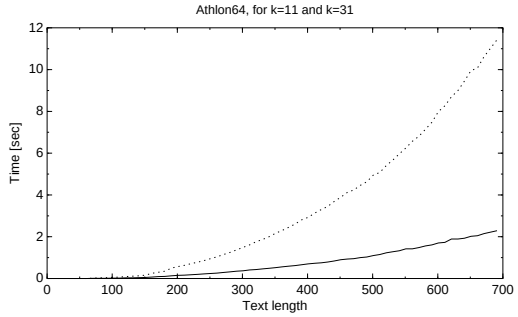

Fig. 3. Time consumption with respect to the text size (solid line for $k = 11$, dotted one for $k = 31$).

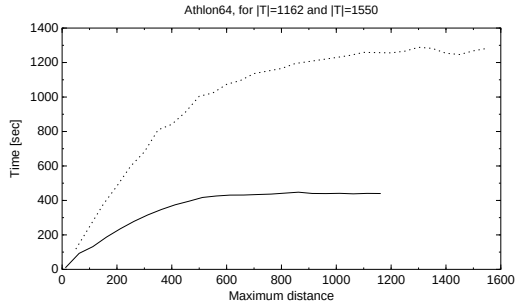

Fig. 4. Time consumption with respect to the distance (solid line for $|T| = 1162$, dotted one for $|T| = 1550$).

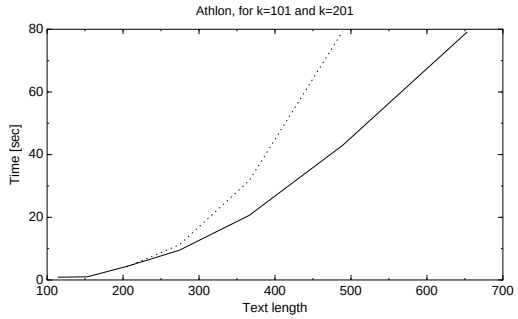

Fig. 5. Time consumption with respect to the text size (solid line for $k = 101$, dotted one for $k = 201$).

covers of a string problem for Hamming distance. The presented algorithm is straightforward, easy to understand and to implement and its theoretical and experimental time requirements are comparable to the existing approach ([2]).

The algorithm may be extended to work with other distance functions, possibly using the idea presented in [3]. Theoretical and experimental analysis similar to one presented here may be accomplished. The algorithm may be also extended to use parallelism.

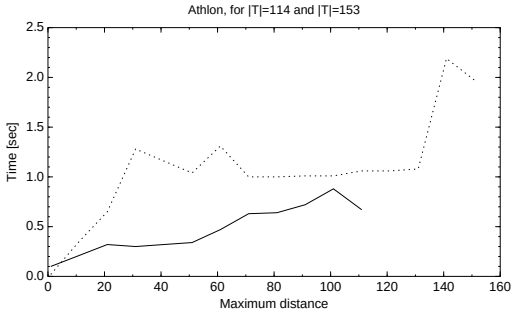

Fig. 6. Time consumption with respect to the maximum distance (solid line for $|T| = 114$, dotted one for $|T| = 153$).

References

1. Apostolico, A., Farach, M., and Iliopoulos, C. S.: Optimal superprimitivity testing for strings. *Inf. Process. Lett. 39*, 1 (1991), 17–20.
2. Christodoulakis, M., Iliopoulos, C. S., Park, K., and Sim, J. S.: Implementing approximate regularities. *Mathematical and Computer Modelling 42* (October 2005), 855–866.
3. Guth, O.: Searching approximate covers of strings using finite automata. In *Proceedings of POSTER* (2008), Faculty of Electrical Engineering, Czech Technical University in Prague.
4. Smyth, W. F.: Approximate periodicity in strings. *Utilitas Mathematica 51* (1997), 125–135.
5. Voráček, M., and Melichar, B.: Searchig for regularities in generalized strings using finite automata. In *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics* (2005), WILEY – VCH Verlag, pp. 809–812.

Measures of quality of rulesets extracted from data*

Martin Holeňa

Institute of Computer Science, Academy of Sciences of the Czech Republic,
Pod Vodárenskou věží 2, 18207 Praha 8, Czech Republic
martin@cs.cas.cz, web:cs.cas.cz/~martin

Abstract. *The paper deals with quality measures of whole sets of rules extracted from data, as a counterpart to more commonly used measures of individual rules. This research has been motivated by increasingly frequent extraction of non-classification rules, such as association rules and rules of observational logic, in real-world data mining tasks. The paer sketches the typology of rules extraction methods and of their rulesets, and recalls that quality measures for whole sets of rules have been so far used only in the case of classification rulesets. It then proposes three possible ways how such measures can be extended to general rulesets. The paper also recalls the possibility to measure the dependence of classification ruleset on parameters of the classification method by means of ROC curves, and proposes a generalization of ROC curves to general rulesets. Finally, a brief illustration on rulesets extracted by means of the method GUHA is given.*

1 Introduction

Logical formulas of specific kinds, usually called *rules*, are a traditional way of formally representing knowledge. Therefore, it is not surprising that they are also the most frequent representation of the knowledge discovered in data mining. Existing methods for rules extraction are based on a broad variety of paradigms and theoretical principles. However, methods relying on different underlying assumptions can lead to the extraction of different or even contradictory rulesets from the same data. Moreover, the set of rules extracted with a particular method can substantially depend on some tunable parameter or parameters of the method, such as significance level, thresholds, size parameters, trade-off coefficients etc. For that reason, it is desirable to have measures of various qualitative aspects of the extracted rulesets. So far, such measures are available only for sets of classification rules, and their dependence on tunable parameters can be described only for classification into two classes [10, 15]. As far as more general kinds of rules are concerned, measures of quality have been proposed only for individual rules [6, 11, 24, 26, 29], or for contrast sets of rules, which finally can be replaced with a single rule [2, 16]; if a whole ruleset is taken into consideration, then only as a context for measuring the quality of an individual rule [27, 28].

* The research reported in this paper has been supported by the grant No. 201/08/1744 of the Grant Agency of the Czech Republic and partially supported by the Institutional Research Plan AV0Z10300504.

The research reporeted in this paper has been motivated by increasingly frequent extraction of non-classification rules in real-world data mining tasks. The paper discusses three possible ways of extending existing ruleset quality measures from classification to general rulesets. The proposed extensions are introduced in Section 4, after the basic typology of rules extraction methods and examples of measures for classification rulesets are recalled in the following two sections, and before a generalization of ROC curves is proposed in Section 5. The paper concludes with a brief illustration on rulesets extracted with the method GUHA.

2 Typology of rules extraction methods

The most natural base for differentiating between existing rules extraction methods is the *syntax and semantics of the extracted rules*. Syntactical differences between them are, however, not very deep since principally, any rule r has one of the forms $S_r \sim S'_r$, or $A_r \rightarrow C_r$, where S_r, S'_r, A_r and C_r are formulas of the considered logic, and $\sim$, $\rightarrow$ are symbols of the language of that logic. The difference between both forms concerns semantic properties of the symbols $\sim$ and $\rightarrow$: $S_r \sim S'_r$ is symmetric with respect to S_r, S'_r in the sense that its validity always coincides with that of $S_r \sim S'_r$ whereas $A_r \rightarrow C_r$ is not symmetric with respect to A_r, C_r in that sense. In the case of a propositional logic, $\sim$ and $\rightarrow$ are the connectives equivalence and implication, respectively, whereas in the case of a predicate logic, they are generalized quantifiers. To distinguish the formulas involved in the asymmetric case, A_r is called *antecedent* and C_r *consequent* of r.

The more important is the semantic of the rules (cf. [6]), especially the difference between *rules of the Boolean logic* and *rules of a fuzzy logic*. Due to the semantics of Boolean and fuzzy formulas, the former are valid for crisp sets of objects, whereas the validity of the latter is a fuzzy set on the universe of all considered objects. Boolean rulesets are extracted more frequently, especially some specific types of them, such as *classification rulesets* [11, 15]. Those are sets of implications such that $(A_r)_{r\in\mathcal{R}}$ and $\{C_r\}_{r\in\mathcal{R}}$ partition the set $\mathcal{O}$ of considered objects, where $\mathcal{R}$ is the considered ruleset, and $\{C_r\}_{r\in\mathcal{R}}$ stands for the set of distinct formulas in $(C_r)_{r\in\mathcal{R}}$. Abandoning the requirement that $(A_r)_{r\in\mathcal{R}}$ partitions $\mathcal{O}$ (at least in the sense of a crisp partitioning) allows to generalize

those rulesets also to fuzzy antecedents. For Boolean antecedents, however, this requirement entails a natural definition of the validity of a whole classification ruleset $\mathcal{R}$ for an object x. Assuming that all information about x conveyed by $\mathcal{R}$ is conveyed by the single rule r covering x (i.e., with A_r valid for x), the validity of $\mathcal{R}$ for x can be defined to coincide with the validity of $A_r \to C_r$ for that r, which in turn equals the validity of C_r for x.

As far as the Boolean predicate logic is concerned, generalized quantifiers both for symmetric and for asymmetric rules were studied in the 1970s within the framework of the *observational logic* [13], which is a Boolean predicate logic with generalized quantifiers. For a set of data about n objects, the truth evaluation of the Boolean predicate φ on those objects is a vector $\|\varphi\| \in \{0,1\}^n$, whereas the truth evaluation of a sentence $(Qx)(\varphi_1(x),\ldots,\varphi_m(x))$ consisting of m Boolean predicates $\varphi_1,\ldots,\varphi_m$ and an m-ary generalized quantifier Q is the function value

$$\|(Qx)(\varphi_1(x),\ldots,\varphi_m(x))\| = \mathrm{Tf}_Q(\|\varphi_1\|,\ldots,\|\varphi_m\|), \tag{1}$$

of a $\{0,1\}$-valued function Tf_Q on the set of m-column binary matrices, which is called *truth function* of the quantifier Q. Observational logic underlies one of the earliest methods for the extraction of general rules from data, called General Unary Hypotheses Automaton (GUHA). In GUHA, the truth function Tf_Q of a generalized quantifier Q is always a function of the 4-fold table

$$\begin{array}{cc|cc} & & S'_r & \neg S'_r \\ & & C_r & \neg C_r \\ \hline S_r & A_r & a & b \\ \neg S_r & \neg A_r & c & d \end{array}. \tag{2}$$

Hence, Tf_Q is a $\{0,1\}$-valued function on quadruples of nonnegative integers. For symmetric rules, GUHA uses quantifiers fulfilling

$$\begin{aligned} &a' \geq a \;\&\; b' \leq b \;\&\; c' \leq c \;\&\; d' \geq d \;\& \\ &\quad\& \;\mathrm{Tf}_Q(a,b,c,d) = 1 \to \mathrm{Tf}_Q(a',b',c',d') = 1. \end{aligned} \tag{3}$$

They are called *associational quantifiers*. For asymmetric rules, it uses quantifiers fulfilling the stronger condition

$$\begin{aligned} &a' \geq a \;\&\; b' \leq b \;\& \\ &\quad\& \;\mathrm{Tf}_Q(a,b,c,d) = 1 \to \mathrm{Tf}_Q(a',b',c',d') = 1. \end{aligned} \tag{4}$$

which are called *implicational quantifiers*. This condition covers also the frequently encountered *association rules* [1, 6, 40] (since methods for the extraction of association rules have been developed outside the framework of observational logic, the terminology is a bit confusing here: although associational rules are asymmetric, their name evokes the quantifier for the symmetric ones).

Orthogonally to the typology according to the semantics of the extracted rules, all extraction methods can be divided into two large groups:

- Methods that extract logical rules from data *directly*, without any intermediate formal representation of the discovered knowledge. Such methods have always formed the mainstream of the extraction of Boolean rules: from the observational logic methods [13] and the method *AQ* [30, 31] in the late 1970s, through the extraction of association rules [1, 40] and the method *CN2* [4], relying on a paradigm similar to that of AQ, to recent methods based on *inductive logic programming* [5, 33] and *genetic algorithms* [9]. They include also important methods for fuzzy rules, in particular *ANFIS* [22, 23] and *NEFCLASS* [34, 35], *fuzzy generalizations* of *observational logic* [18, 19] and a recent method based on *fuzzy transform* [36].
- Methods that employ some *intermediate representation* of the extracted knowledge, useful by itself. This group includes two important kinds of methods: *classification trees* [3, 37] and methods based on *artificial neural networks (ANN)*. The latter are used both for Boolean and for fuzzy rules [7, 21, 39] (cf. also the survey papers [32, 38]).

3 Existing measures for classification rulesets

A survey of measures of quality for classification rulesets (with possibly fuzzy antecedents) has been given in the monograph [15]. All measures have been divided there into four groups: inaccuracy, imprecision, inseparability and resemblance. Space limitation allows to recall here only the main representatives of the more important groups:

Inaccuracy measures the discrepancy between the true class of the considered objects and the class predicted by the ruleset. Its most frequently encountered representative is the *quadratic score* (also called Brier score):

$$\mathrm{Inacc} = \frac{1}{|\mathcal{O}|} \sum_{x\in\mathcal{O}} \sum_{C\in\{C_r\}_{r\in\mathcal{R}}} \Big(\delta_C(x) - \hat{\delta}_C(x)\Big)^2, \tag{5}$$

where $|\ |$ denotes cardinality, $\mathcal{O}$ is the considered set of objects, $\delta_C(x) \in \{0,1\}$ is the validity of the proposition C for $x \in \mathcal{O}$, and $\hat{\delta}_C(x)$ is the agreement between C and the class predicted for x by $\mathcal{R}$. In the general case of a fuzzy logic, $\hat{\delta}_C(x) = \max_{C_r=C}\|A_r\|_x$, with $\|A_r\|_x \in \langle 0,1\rangle$ denoting the truth grade of A_r for x.

Imprecision measures the discrepancy between the probability distribution of the classes, conditioned on the values of attributes occurring in antecedents, and the class predicted by the ruleset. Its most common representative is

$$\begin{aligned} &\mathrm{Impr} = \\ &= \frac{1}{|\mathcal{O}|} \sum_{x\in\mathcal{O}} \sum_{C\in\{C_r\}_{r\in\mathcal{R}}} \Big(\delta_C(x) - \hat{\delta}_C(x)\Big)\Big(1 - \hat{\delta}_C(x)\Big)^2. \end{aligned} \tag{6}$$

As was already mentioned in the introduction, the extracted ruleset can substantially depend on tunable parameters of the employed method. This was so far systematically studied only for dichotomous classification with $\mathcal{R} = \{A \to C, \neg A \to \neg C\}$. In that case, putting $A_r = A$, $C_r = C$ allows the information about the validity of A and C for $\mathcal{O}$ to be again summarized by means of the 4-fold table (2), which also depends on the parameter values. The influence of the parameter values on the result of dichotomous classification is usually investigated by means of the measures *sensitivity* $= \frac{a}{a+c}$ and *specificity* $= \frac{d}{b+d}$ [15]. Connecting points (1-specificity,sensitivity) $= (\frac{b}{b+d}, \frac{a}{a+c})$ for the considered parameter values forms a curve with graph in the unit square, called *receiver operating characteristic* (ROC), due to the area where such curves have first been in routine use. In machine learning, a modified version of those curves has been proposed, in which the points connected for considered parameter values are (b, a) [10]. The graph of such a curve then lies in the rectangle with vertices $(0, 0)$ and $(b + d, a + c)$, and is called *coverage graph*.

The graphs of ROC curves and coverage graphs can provide information about the influence of parameter values not only on the sensitivity and specificity, but also on other measures. It is sufficient to complement the graph with isolines of the measure and to investigate their intersections with the original curve [10].

4 Three extensions to more general kinds of rules

In the particular case of classification rulesets with Boolean antecedents, some algebra allows to substantially simplify (5)–(6):

$$\begin{aligned} \mathrm{Inacc} &= \frac{2|\mathcal{O}^-|}{|\mathcal{O}|} = 1 - \frac{|\mathcal{O}^+| - |\mathcal{O}^-|}{|\mathcal{O}|}, \\ \mathrm{Impr} &= \frac{|\mathcal{O}^-|}{|\mathcal{O}|} = 1 - \frac{|\mathcal{O}^+|}{|\mathcal{O}|}, \end{aligned} \tag{7}$$

where

$$\begin{aligned} \mathcal{O}^+ &= \{x \in \mathcal{O} : \mathcal{R} \text{ is valid for } x\}, \\ \mathcal{O}^- &= \{x \in \mathcal{O} : \mathcal{R} \text{ is not valid for } x\}. \end{aligned} \tag{8}$$

This not only shows that, in the case of Boolean antecedents, the quadratic score is sufficient to describe also the imprecision, but also suggests an approach how to extend those measures to general rulesets: to use (7)–(8) as the definition of measures (5)–(6). More generally, any measure of quality of classification rulesets with Boolean antecedents (e.g., any measure surveyed in [15]) that can be reformulated by means of $\mathcal{O}^+$ and $\mathcal{O}^-$, can be extended in such a way that the reformulation is used as the definition of that measure for general rulesets.

For sets of asymmetric rules, also the notion of covering an object by a rule, which was recalled in Section 2, can be generalized. Notice, however, that for fuzzy antecedents, the validity of A_r, $r \in \mathcal{R}$ is a fuzzy set on $\mathcal{O}$. Consequently, the set $\mathcal{O}_\mathcal{R}$ of objects covered by $\mathcal{R}$ is a fuzzy set on $\mathcal{O}$ with the membership function

$$\mu_\mathcal{R}(x) = \|(\exists r \in \mathcal{R})\ A_r\|_x = \max_{r \in \mathcal{R}} \|A_r\|_x. \tag{9}$$

Observe that according to (9), $\mathcal{O}_\mathcal{R} = \mathcal{O}$ for classification rulesets with Boolean antecedents. Therefore, various generalizations of classification measures to general rulesets of asymmetric rules are possible: wherever $\mathcal{O}$ occurs in the definition of a measure for classification rulesets, either $\mathcal{O}$ or $\mathcal{O}_\mathcal{R}$ can occur in its general definition, provided $\mathcal{O}_\mathcal{R} \neq \emptyset$. To allow unified treatment of symmetric and asymmetric rules, the concept of covering an object by a rule will be extended also to symmetric rules, in such a way that an object x is covered by $S_r \sim S'_r$ if either S_r or S'_r is valid for x. Hence, a counterpart of (9) for a set $\mathcal{R}$ is a fuzzy set with the membership function

$$\begin{aligned} \mu_\mathcal{R}(x) = \|(\exists r \in \mathcal{R})(S_r \vee S'_r)\|_x = \\ = \max_{r \in \mathcal{R}} \max(\|S_r\|_x, \|S'_r\|_x). \end{aligned} \tag{10}$$

According to (8), the proposed way of extending measures of quality from classification rulesets with Boolean antecedents to general rulesets requires to generalize the concept of validity of a general ruleset for an object. However, there are multiple possibilities for such a generalization. Indeed, at least any of the following points of view is possible:

Boolean validity of the ruleset based on simultaneous validity of all covering rules. According to this point of view, the validity of a ruleset $\mathcal{R}$ for a covered object x is a Boolean property expressing the simultaneous validity of all rules that cover x. Consequently, the sets $\mathcal{O}^+$ and $\mathcal{O}^-$ defined in (8) are crisp sets

$$\begin{aligned} \mathcal{O}^+ = \{x \in \mathcal{O} : \mu_\mathcal{R}(x) > 0\ \& \\ (\forall r \in \mathcal{R})\ \|r \text{ covers } x\ \&\ r \text{ is valid for } x\| = \|r \text{ covers } x\|\}, \end{aligned} \tag{11}$$

$$\begin{aligned} \mathcal{O}^- = \{x \in \mathcal{O} : \mu_\mathcal{R}(x) > 0\ \& \\ (\exists r \in \mathcal{R})\ \|r \text{ covers } x\ \&\ r \text{ is valid for } x\| < \|r \text{ covers } x\|\}, \end{aligned} \tag{12}$$

where

$$\|r \text{ covers } x\| = \begin{cases} \|(S_r \vee S'_r)\|_x & \text{for symmetric rules ,} \\ \|A_r\|_x & \text{for asymmetric rules ,} \end{cases} \tag{13}$$

and similarly

$$\begin{aligned} \|r \text{ covers } x\ \&\ r \text{ is valid for } x\| = \\ = \begin{cases} \|(S_r \vee S'_r)\&r\|_x & \text{for symmetric rules ,} \\ \|A_r\&r\|_x & \text{for asymmetric rules .} \end{cases} \end{aligned} \tag{14}$$

The following consequences of this point of view are worth noticing:

(i) It is immaterial how the truth grade $\|r\|_x$ of a rule r being valid for an object x is evaluated (thus also how $\|\neg r\|_x$ is evaluated).

(ii) If $\mu_{\mathcal{R}}(x) = 0$, then $x \notin \mathcal{O}^+ \cup \mathcal{O}^-$.

(iii) For classification rulesets with Boolean antecedents, the validity of $\mathcal{R}$ according to this point of view coincides with the definition in Section 2 because in that case, there is exactly one rule that covers x.

Boolean validity of the ruleset based on the validity of the majority of covering rules. According to this point of view, the validity of a ruleset $\mathcal{R}$ for a covered object x is a Boolean property expressing the validity of most of the rules that cover x. Consequently, the sets $\mathcal{O}^+$ and $\mathcal{O}^-$ in (8) are crisp sets

$$\begin{aligned}\mathcal{O}^+ = \{x \in \mathcal{O} : \mu_{\mathcal{R}}(x) > 0 \;\&\\ \&\; \sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; r \text{ is valid for } x\| >\\ > \sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; \neg r \text{ is valid for } x\|\}, \end{aligned} \quad (15)$$

$$\begin{aligned}\mathcal{O}^- = \{x \in \mathcal{O} : \mu_{\mathcal{R}}(x) > 0 \;\&\\ \&\; \sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; r \text{ is valid for } x\|\\ \leq |\sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; \neg r \text{ is valid for } x\|\}, \end{aligned} \quad (16)$$

where the truth grade $\|r \text{ covers } \;\&\; \neg r \text{ is valid for } x\|$ is again evaluated according to (14), replacing r with $\neg r$. Observe that also this point of view has the above consequences (i)–(iii), the last one again due to the fact that there is exactly one rule covering x.

Fuzzy validity of the ruleset based on the relative validity of covering rules. In this case, the validity of a ruleset $\mathcal{R}$ for a covered object x is a fuzzy property expressing the ratio of the validity of rules from $\mathcal{R}$ for x to the covering of x with those rules. Consequently, the sets $\mathcal{O}^+$ and $\mathcal{O}^-$ are fuzzy sets on $\mathcal{O}$ with memberships μ_+ and μ_-, respectively, such that if $\mu_{\mathcal{R}}(x) > 0$,

$$\mu_+(x) = \frac{\sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; r \text{ is valid for } x\|}{\sum_{r\in\mathcal{R}} \|r \text{ covers } x\|\|} \quad (17)$$

$$\mu_-(x) = \frac{\sum_{r\in\mathcal{R}} \|r \text{ covers } x \;\&\; \neg r \text{ is valid for } x\|}{\sum_{r\in\mathcal{R}} \|r \text{ covers } x\|\|} \quad (18)$$

where the involved truth grades are again evaluated according to (13) and (14). Moreover, (17)–(18) will be complemented with the definition $\mu_+(x) = \mu_-(x) = 0$ if $\mu_{\mathcal{R}}(x) = 0$, to get again the validity of (ii) above, whereas (i) and (iii) are consequences also of this point of view. Further, the fact that $\mathcal{O}^+$ and $\mathcal{O}^-$ are now fuzzy sets implies that whenever $|\mathcal{O}^+|$ or $|\mathcal{O}^-|$ occur in the definitions of quality measures for Boolean classification rulesets, fuzzy cardinalities have to be used in their generalizations to general rulesets according to this point of view. Hence,

$$|\mathcal{O}^+| = \sum_{x\in\mathcal{O}} \mu_+(x),\ |\mathcal{O}^-| = \sum_{x\in\mathcal{O}} \mu_-(x). \quad (19)$$

For example, the measure

$$\text{Inacc} = 1 - \frac{\sum_{x\in\mathcal{O}} (\mu_+(x) - \mu_-(x))}{|\mathcal{O}|} \quad (20)$$

is a generalization of (5), whereas the measures

$$\text{Impr}_1 = 1 - \frac{\sum_{x\in\mathcal{O}} \mu_+(x)}{|\mathcal{O}|}, \quad (21)$$

$$\text{Impr}_2 = 1 - \frac{\sum_{x\in\mathcal{O}} \mu_+(x)}{|\mathcal{O}_{\mathcal{R}}|} = 1 - \frac{\sum_{x\in\mathcal{O}} \mu_+(x)}{\sum_{x\in\mathcal{O}} \mu_{\mathcal{R}}(x)} \quad (22)$$

are generalizations of (6).

5 Extensions of ROC curves to more general kinds of rules

Observe that in the case of Boolean classification with $\mathcal{R} = \{A \to C, \neg A \to \neg C\}$, the information about the validity of $\mathcal{R}$ for objects $x \in \mathcal{O}$ can be also viewed as information about the validity of a ruleset $\mathcal{R}' = \{A \to C\}$. However, $\mathcal{R}'$ is not any more a classification ruleset, but only a general one, which can be described only by means of the above introduced sets $\mathcal{O}_{\mathcal{R}}$, $\mathcal{O}^+$, $\mathcal{O}^-$. In particular, $|\mathcal{O}^+| = a$ and $|\mathcal{O}^-| = b$, which suggests the possibility to generalize coverage graphs introduced in Section 3 to general rulesets by means of a curve connecting points $(|\mathcal{O}^-|, |\mathcal{O}^+|)$ for each of the values of the considered parameters. For a generalization of ROC curves to general rulesets, those points have to be scaled to the unit square. Since the resulting curve will be used to investigate the dependence on parameter values, the scaling factor itself must be independent of those values. The only available factor fulfilling this condition is the number of objects, $|\mathcal{O}|$ (the other available factors, $|\mathcal{O}_{\mathcal{R}}|$, $|\mathcal{O}^+|$ and $|\mathcal{O}^-|$ depend on the evaluations $\|S_r\|$ and $\|S'_r\|$, or $\|A_r\|$ and $\|C_r\|$, which in turn depend on the parameter values). Consequently, the proposed generalization of ROC curves will connect points $(\frac{|\mathcal{O}^-|}{|\mathcal{O}|}, \frac{|\mathcal{O}^+|}{|\mathcal{O}|})$.

For practical construction of the proposed generalization of ROC curves, the following proposition, proven in [17], can be quite useful:

Proposition 1. *Let the covering of individual objects with individual rules be a Boolean property (i.e., the set of rules covering a particular object x be a crisp subset of $\mathcal{R}$). Then irrespectively of which of the above points of view of ruleset validity is adopted, there always exists a constant $c \in \langle 0,1\rangle$ and an increasing bijection $g : \langle 0,c\rangle \to \langle 0,1\rangle$ such that*

$$|\mathcal{O}^+| + |\mathcal{O}^-| \leq \max(1, \max_{x\in\langle 0,c\rangle} x + g^{-1}(1-g(x)))|\mathcal{O}|. \tag{23}$$

Moreover, in the particular cases of Boolean logic and of all three fundamental fuzzy logics (Łukasiewicz, Gödel, product), (23) holds with $c = 1$ and g equal to identity,

$$|\mathcal{O}^+| + |\mathcal{O}^-| \leq |\mathcal{O}|. \tag{24}$$

Thus in those cases, the points $(\frac{|\mathcal{O}^-|}{|\mathcal{O}|}, \frac{|\mathcal{O}^+|}{|\mathcal{O}|})$, forming the generalization of ROC curves, lie below the diagonal $(\langle 0,1\rangle, \langle 1,0\rangle)$.

The proposition is illustrated in Figure 1, together with isolines of the three example measures introduced in (20)–(22). Observe that the isolines of Impr_2 depend on the relationship between the three cardinalities $|\mathcal{O}^+| = \sum_{x\in\mathcal{O}} \mu_+(x)$, $|\mathcal{O}^-| = \sum_{x\in\mathcal{O}} \mu_-(x)$ and $|\mathcal{O}_\mathcal{R}| = \sum_{x\in\mathcal{O}} \mu_\mathcal{R}(x)$. The isolines depicted in Figure 1(c) correspond to the relationship $|\mathcal{O}_\mathcal{R}| = |\mathcal{O}^+| + |\mathcal{O}^-|$, which is true in Łukasiewicz logic (thus in particular also in Boolean logic).

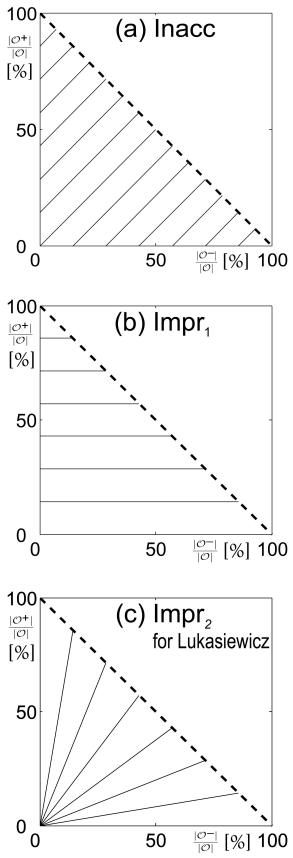

Fig. 1. Isolines of the three measures introduced in (20)–(22), drawn with respect to the coordinates $(\frac{|\mathcal{O}^-|}{|\mathcal{O}|}, \frac{|\mathcal{O}^+|}{|\mathcal{O}|})$ of points forming the proposed generalization of ROC curves.

6 Experimentally testing the approach

The proposed approach has been so far experimentally tested for six rules extraction methods on three benchmark data sets, as well as on data from one real-world knowledge discovery task [20]. For each method, 1–3 parameters were tuned, the values of them being chosen among 2–10 possibilities. For some data sets, some combinations of parameter values did not extract any rules. Whenever a particular combination of parameter vaules extracted a nonempty ruleset from the considered data, it was tested on those data by means of a 10-fold crossvalidation. Consequently, the number of rulesets extracted from each data set varied between 1000 and 1500.

As a very brief illustration, Figure 2 shows the proposed generalization of ROC curves for two rulesets extracted from the best known benchmark set, the iris data, originally used in 1930s by R.A. Fisher [8], by means of the GUHA quantifier *founded implication*. This quantifier, denoted $\to_{s,\theta}$, $s, \theta \in \langle 0,1\rangle$ has its truth function $\mathrm{Tf}_{\to_{s,\theta}}$ defined in such a way that the rule $A_r \to_{s,\theta} C_r$ is valid exactly for those data for which the conditional probability $p(C_r|A_r)$ of the validity of C_r conditioned on A_r, estimated with the unbiased estimate $\frac{a}{a+b}$, is at least θ, whereas A_r and C_r are simultaneously valid in at least the proportion s of the data [13]. Hence, $\mathrm{Tf}_{\to_{s,\theta}} = 1$ iff $\frac{a}{a+b} \geq \theta$ & $\frac{a}{a+b+c+d} \geq s$. As was pointed out in [14], rules with this quantifier are actually association rules with support s and confidence θ. Each curve corresponds to changing only one of the parameters s, θ, the value of the other is fixed.

7 Conclusions

The paper has dealt with quality measures of rules extracted from data, though not in the usual context of individual rules, but in the context of whole rulesets. Three kinds of extensions of measures already in use for classification rulesets have been proposed. In addition, the concept of ROC-curves has been generalized, to enable investigating the dependence of general rulesets on the values of parameters of the extraction method.

The paper actuallly discusses some general aspects related to an ongoing investigation into the possibility to reflect uncertain validity of rulesets extracted from data when measuring their quality. The outcomes of that investigation

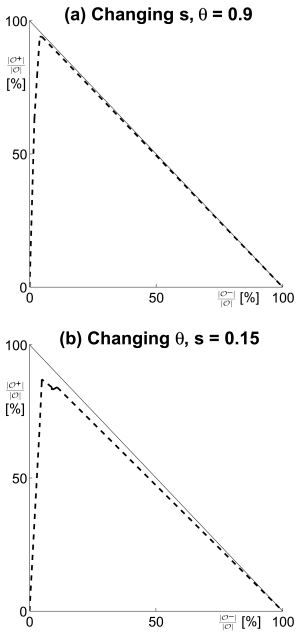

Fig. 2. Example of generalized ROC curves for rulesets extracted from the iris data by means of the GUHA quantifier founded implication.

are intended to be published elsewhere [17]. They comprise theoretical elaboration of the last proposed kind of extensions of ruleset quality measures, as well as results of extensive experimental tests on rulesets extracted from benchmark and real-world data sets by means of six methods attempting to cover a possibly broad spectrum of rules extraction methods. Those results indicate that the approach is feasible and can contribute to the ultimate objective of quality measures: to allow comparing the knowledge extracted with different data mining methods and investigating how the extracted knowledge depends on the values of their parameters.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, 1996.
2. S.D. Bay and M.J. Pazzani. Detecting group differences. mining contrast sets. *Data Mining and Knowledge Discovery*, 5:213–246, 2001.
3. L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.
4. P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Machine Learning – EWSL-91*, pages 151–163. Springer Verlag, New York, 1991.
5. L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London, 1992.
6. D. Dubois, Hüllermeier, and H. Prade. A systematic approach to the assessment of fuzzy association rules. *Data Mining and Knowledge Discovery*, 13:167–192, 2006.
7. W. Duch, R. Adamczak, and K. Grabczewski. A new methodology of extraction, optimization and application of crisp and fuzzy logical rules. *IEEE Transactions on Neural Networks*, 11:277–306, 2000.
8. R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7:179–188, 1936.
9. A.A. Freitas. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer Verlag, Berlin, 2002.
10. J. Fürnkranz and P.A. Flach. ROC 'n' rule learning – towards a better understanding of covering algorithms. *Machine Learning*, 58:39–77, 2005.
11. L. Geng and H.J. Hamilton. Choosing the right lens: Finding what is interesting in data mining. In F. Guillet and H.J. Hamilton, editors, *Quality Measures in Data Mining*, pages 3–24. Springer Verlag, Berlin, 2007.
12. P. Hájek. *Metamathematics of Fuzzy Logic*. Kluwer Academic Publishers, Dordrecht, 1998.
13. P. Hájek and T. Havránek. *Mechanizing Hypothesis Formation*. Springer Verlag, Berlin, 1978.
14. P. Hájek and M. Holeňa. Formal logics of discovery and hypothesis formation by machine. *Theoretical Computer Science*, 292:345–357, 2003.
15. D.J. Hand. *Construction and Assessment of Classification Rules*. John Wiley and Sons, New York, 1997.
16. R.J. Hilderman and T. Peckham. Statistical methodologies for mining potentially interesting contrast sets. In F. Guillet and H.J. Hamilton, editors, *Quality Measures in Data Mining*, pages 153–177. Springer Verlag, Berlin, 2007.
17. M. Holeňa. Measures of ruleset quality capable to represent uncertain validity. Submitted to *International Journal of Approximate Reasoning*.
18. M. Holeňa. Fuzzy hypotheses for Guha implications. *Fuzzy Sets and Systems*, 98:101–125, 1998.
19. M. Holeňa. Fuzzy hypotheses testing in the framework of fuzzy logic. *Fuzzy Sets and Systems*, 145:229–252, 2004.
20. M. Holeňa. Neural networks for extraction of fuzzy logic rules with application to EEG data. In B. Ribeiro, R.F. Albrecht, and A. Dobnikar, editors, *Adaptive and Natural Computing Algorithms*, pages 369–372. Springer Verlag, Wien, 2005.
21. M. Holeňa. Piecewise-linear neural networks and their relationship to rule extraction from data. *Neural Computation*, 18:2813–2853, 2006.
22. J.S.R. Jang. ANFIS: Adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man, and Cybernetics*, 23:665–685, 1993.
23. J.S.R. Jang and C.T. Sun. Neuro-fuzzy modeling and control. *The Proceedings of the IEEE*, 83:378–406, 1995.
24. K.A. Kaufman and R.S. Michalski. An adjustable description quality measure for pattern discovery using the AQ methodology. *Journal of Intelligent Information Systems*, 14:199–216, 2000.
25. E.P. Klement, R. Mesiar, and E. Pap. *Triangular Norms*. Kluwer Academic Publishers, Dordrecht, 2000.
26. S. Lallich, O. Teytaud, and E. Prudhomme. Association rule interestingness: Measure and statistical validation. In F. Guillet and H.J. Hamilton, editors, *Quality Measures in Data Mining*, pages 251–275. Springer Verlag, Berlin, 2007.

27. P. Lenca, B. Vaiilant, P. Meyer, and S. Lalich. Association rule interestingness meaures: Experimental and theoretical studies. In F. Guillet and H.J. Hamilton, editors, *Quality Measures in Data Mining*, pages 51–76. Springer Verlag, Berlin, 2007.
28. L. Lerman and J. Azè. Une mesure probabiliste contextuelle discriminante de qualite des règles d'association. In *EGC 2003: Extraction et Gestion des Connaissances*, pages 247–263. Hermes Science Publications, Lavoisier, 2003.
29. K. McGarry. A survey of interestingness measures for knowledge discovery. *Knowledge Engineering Review*, 20:39–61, 2005.
30. R.S. Michalski. Knowledge acquisition through conceptual clustering: A theoretical framework and algorithm for partitioning data into conjunctive concepts. *International Journal of Policy Analysis and Information Systems*, 4:219–243, 1980.
31. R.S. Michalski and K.A. Kaufman. Learning patterns in noisy data. In *Machine Learning and Its Applications*, pages 22–38. Springer Verlag, New York, 2001.
32. S. Mitra and Y. Hayashi. Neuro-fuzzy rule generation: Survey in soft computing framework. *IEEE Transactions on Neural Networks*, 11:748–768, 2000.
33. S. Muggleton. *Inductive Logic Programming*. Academic Press, London, 1992.
34. D. Nauck. Fuzzy data analysis with NEFCLASS. *International Journal of Approximate Reasoning*, 32:103–130, 2002.
35. D. Nauck and R. Kruse. NEFCLASS-X: A neuro-fuzzy tool to build readable fuzzy classifiers. *BT Technology Journal*, 3:180–192, 1998.
36. V. Novák, I. Perfilieva, A. Dvořák, C.Q. Chen, Q. Wei, and P. Yan. Mining pure linguistic associations from numerical data. To appear in International Journal of Approximate Reasoning.
37. J. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Francisco, 1992.
38. A.B. Tickle, R. Andrews, M. Golea, and J. Diederich. The truth will come to light: Directions and challenges in extracting rules from trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9:1057–1068, 1998.
39. H. Tsukimoto. Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11:333–389, 2000.
40. M.J. Zaki, S. Parathasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery*, 1:343–373, 1997.

The coin flipping selector for selective encryption*

Richard Ostertág

Department of Computer Science,
Faculty of Mathematics, Physics and Informatics, Comenius University,
Mlynská dolina, 842 48 Bratislava, Slovak Republic
ostertag@dcs.fmph.uniba.sk, http://www.dcs.fmph.uniba.sk

Abstract. *Some applications require high-speed encryption even at the expense of reduced security. With a fixed secure, but slow cryptographic algorithm, there still is an appealing possibility for encryption speedup by encrypting only some portion of data. In this paper we analyze the ciphertext security obtained this way. We show that it is not possible to exclude from encryption even a small constant fraction of data without significantly compromising security.*

1 Motivation, assumptions, goals

Volume of data is nowadays bigger than ever. Multimedia are a typical example. Fast real-time on-demand encryption of multiple multimedia streams requires specialized powerful hardware.

It is sometimes not possible (or economical) to use powerful enough hardware solution. Then we can replace the encryption algorithm with a faster – although maybe less secure one. Another possibility is to use selective encryption with the original secure algorithm. In this case we encrypt only some fraction of plaintext. Let p denote the fraction of encrypted plaintext. The parameter p ranges between 0 (no encryption) and 1 (full encryption) and is used to control the balance between the encryption speedup and the security.

For example, selective encryption is used for on-line encryption of MPEG video [1]. In this case, the knowledge of the internal data structure is exploited in order to encrypt only DC coefficients and sign bits of motion vectors. Similar techniques are also used for pictures [2]. For overview of selective encryption methods see [3]. Security of these algorithms is not formally proved.

We formally analyze security of selective encryption in this paper. As we are interested in a general case, we make no assumptions on the internal data structure or on statistical properties of the plaintext.

We originally hoped that it could be possible to selectively encrypt portion of plaintext while maintaining reasonable security. However, we show that this does not work. Since we prove a negative result, it is only better if assumptions are more disadvantageous for the attacker than in practical usage:

1. *One-time pad is used as the encrypting algorithm.*
   One-time pad is the first and only encryption algorithm for which there is a proof of perfect secrecy if the key is truly random, never reused, and kept secret. We choose this cipher to abstract from eventual weaknesses of the actual cipher which can be exploited by attacker. Theoretical results obtained this way can be used in practice as upper bounds for security of any other selected encryption algorithm.
2. *Attacker can manage no more than ciphertext-only attack.*
   The attacker is assumed to have access only to a ciphertext and full description of selective encryption algorithm. This means that the attacker knows the enciphering algorithm and also the method of bit selection for enciphering.
3. *Attack is peformed using brute force.*
   Key space is searched from the most probable key to the least probable key omitting impossible keys to minimize the attacker's work. We assume that the selection algorithm chooses bits for encrypting independently from plaintext content (besides its length). In general it cannot be expected that a better attack is possible. However in actual situation specific properties[1] of plaintext can lead to a more efficient attack.
4. *Attack complexity measure is defined as a fraction of key space that attacker has to search in average to find the key.*
   Attacker tries every possible key until he finds one that deciphers to the desired plaintext. We ignore the complexity of verifying whether deciphered plaintext is the original one. For selective encryption with $p = 1$ (one-time pad), the expected attack complexity is $1/2$. For selective encryption with $p = 0$ expected complexity is 0. We consider every cipher for which attack complexity approaches 0 as plaintext length goes to $+\infty$ insecure.

We assume that encrypting p percent of plaintext bits with selective encryption reduces sender's work to p percent omitting overhead necessary for selecting those bits. In this situation we will be satisfied with (and accept this as reasonable degradation of security) reduction of attack

* Supported by VEGA grant No. 1/3106/06.

[1] E.g. high redundancy of plaintext poses an even greater risk for selective encryption then for full text encryption.

complexity from $1/2$ to $p/2$, because this means that attacker's work is in average also reduced to p percent but no more.

2 Selectors

In this paper we will assume that plaintext is a bit sequence – sequence of zeros and ones. Let $n > 0$ denote plaintext sequence length. If we want to selectively encrypt $p \in (0, 1)$ percent of plaintext, then we have to choose $k = np$ bits of plaintext for encryption. In [4] we analyzed different ways of bit selection for selective encryption. We introduced the notion of selector – algorithm which performs selection of the k bits for encryption based on n and p. The output of selector on input n and p is a bit sequence of length n with $k = np$ ones – indicating positions of bits chosen for encryption. Selective encryption algorithm proceeds in the following way:

1. The selector selects $k = np$ bits for encryption.
2. Encrypt only selected bits with one-time pad[2].

As it can be seen our model was limited to selections which have exactly $k = np$ bits selected. In [4] we proved that among the analyzed selectors only fully random selection of exactly p percents of bits provides reasonable security for $p \geq 1/2$. In this place it is necessary to mention that in [4] we measured the attack complexity by the number of possible plaintexts[3].

Because we are interested in the values of $p < 1/2$ we relax the assumption that exactly $k = np$ bits have to be selected, and we only require that in average k bits have to be selected. This relaxation allows for using a selector which for every bit flips a biased coin – one falls with probability p, zero with probability $1 - p$. Lets call this selector coin flipping selector. We hope that this step allow us to go with p below $1/2$ because it introduce more uncertainty to attacker as all plaintext are now possible. For that reason we have to also change our attack complexity measure and we choose one mentioned in previous section.

3 The coin flipping selector analysis

In the rest of the paper we will show the behavior of the attack complexity for the coin flipping selector for large messages (we will assume that n goes to infinity).

[2] Xor them with truly random noise.

[3] For example, let $p = 1/2$. For a random bit selector there are 2^{n-1} possible plaintexts for every ciphertext. If the selector do not use randomness and deterministically selects every even bit, there are only $2^{n/2}$ possible plaintexts.

3.1 Average fraction of key space equation

Firstly we need to determine the probability of the key of length n with exactly k ones on fixedly chosen positions if in the selective encryption the coin flipping selector is used. Let denote this probability as $\mathrm{PK}(n, k, p)$, where p is probability of encrypting.

Theorem 1.

$$\mathrm{PK}(n, k, p) = \left(\frac{p}{2}\right)^k \left(1 - \frac{p}{2}\right)^{n-k}$$

Proof.

$$\mathrm{PK}(n, k, p) = \sum_{i=0}^{n-k} \binom{n-k}{i} p^{k+i} (1-p)^{(n-k)-i} \frac{1}{2^{k+i}},$$

because we can get the key with exactly k ones on fixedly chosen positions from any selection with exactly $k+i$ ones with k ones on those fixedly chosen positions and i ones arbitrarily chosen from remaining $n - k$ positions. Also one-time pad has to select for those k positions bit 1 and for remaining i positions bit 0 (thus we get $2^{-(k+i)}$). We can simplify the last equation as follows:

$$\left(\frac{p}{2}\right)^k \sum_{i=0}^{n-k} \binom{n-k}{i} \left(\frac{p}{2}\right)^i (1-p)^{(n-k)-i} =$$

$$= \left(\frac{p}{2}\right)^k \left(\frac{p}{2} + (1-p)\right)^{n-k} = \left(\frac{p}{2}\right)^k \left(1 - \frac{p}{2}\right)^{n-k}.$$

□

Derivative of the function $\mathrm{PK}(n, k, p)$ with respect to k is:

$$\mathrm{PK}(n, k, p) \ln\left(\frac{p}{2-p}\right).$$

Since for all studied $n > 0$ and $0 < p < 1$ expression $\ln(\frac{p}{2-p})$ is negative and $\mathrm{PK}(n, k, p)$ is positive we know, that $\mathrm{PK}(n, k, p)$ is strictly decreasing function with respect to $k \in \langle 0, n \rangle$. Thus effective attacker will start searching key space from the most probable 0^n key to the least probable 1^n key in direction of increasing number of ones in the key. Sort all 2^n keys in this order[4] in an array with indexes from 1 to 2^n. Then denote $L(n, k)$ index of first key of length n with k ones and $U(n, k)$ will denote index of the last key of length n with k ones. It can easily be seen that:

$$L(n, k) = 1 + \sum_{i=0}^{k-1} \binom{n}{i}, U(n, k) = \sum_{i=0}^{k} \binom{n}{i}.$$

[4] Ordering of keys with equal number of ones is irrelevant since all have the same probability. It can be arbitrary but fixed.

Theorem 2. *Let $I(n,p)$ be a position in the above mentioned array where attacker finds the key in average case. Then $I(n,p)$ equals to:*

$$\frac{1}{2}+\frac{1}{2}\left(\frac{2-p}{p}\right)^n\sum_{k=0}^{n}\left[\left(\frac{p}{2-p}\right)^k\binom{n}{k}\sum_{i=0}^{k}\binom{n+1}{i}\right].$$

Proof. Let $\Pr(n,p,i)$ be probability that attacker finds key in position i. Then:

$$I(n,p)=\sum_{i=1}^{2^n} i\Pr(n,p,i).$$

Since $\Pr(n,p,i)$ is constant for all i between $L(n,k)$ and $U(n,k)$ we can write:

$$I(n,p)=\sum_{k=0}^{n}\left[\sum_{i=L(n,k)}^{U(n,k)} i\,\mathrm{PK}(n,k,p)\right]$$

Since $\mathrm{PK}(n,k,p)$ does not depend on i we can move it in front of inner sum. The inner sum then reduces to:

$$\sum_{i=L(n,k)}^{U(n,k)} i=[U(n,k)-L(n,k)+1]\frac{L(n,k)+U(n,k)}{2}$$

Thus equation for $I(n,p)$ changes to:

$$\sum_{k=0}^{n}\mathrm{PK}(n,k,p)\frac{1}{2}\binom{n}{k}\underbrace{\left[1-\binom{n}{k}+2\sum_{i=0}^{k}\binom{n}{i}\right]}_{\text{Mark this term as } x(n,k).}.$$

It is obvious that $x(n,0)=2$ and $x(n,k+1)-x(n,k)=\binom{n+1}{k+1}$. So $x(n,k)=1+\sum_{i=0}^{k}\binom{n+1}{i}$. After substituting $x(n,k)$ and $\mathrm{PK}(n,k,p)$ we can write $I(n,p)$ as:

$$\frac{1}{2}\sum_{k=0}^{n}\left(\frac{p}{2}\right)^k\left(1-\frac{p}{2}\right)^{n-k}\binom{n}{k}\left[1+\sum_{i=0}^{k}\binom{n+1}{i}\right].$$

Then after factoring out $\left(1-\frac{p}{2}\right)^n$ we get:

$$\frac{1}{2}\left(1-\frac{p}{2}\right)^n\sum_{k=0}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\left[1+\sum_{i=0}^{k}\binom{n+1}{i}\right].$$

By expanding summand and using binomial theorem for $\left(\frac{p}{2-p}+1\right)^n$ we get:

$$\frac{1}{2}\left(1-\frac{p}{2}\right)^n\left(\frac{2}{2-p}\right)^n+$$

$$+\frac{1}{2}\left(1-\frac{p}{2}\right)^n\sum_{k=0}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\sum_{i=0}^{k}\binom{n+1}{i}=$$

$$=\frac{1}{2}+\frac{1}{2}\left(\frac{2-p}{2}\right)^n\sum_{k=0}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\sum_{i=0}^{k}\binom{n+1}{i}.$$

□

Let us denote average fraction of key space which attacker has to search before he finds the key as $F(n,p)$. Now, when we have $I(n,p)$, equation for F is obvious:

$$F(n,p)=\frac{\frac{1}{2}+\frac{1}{2}\left(\frac{2-p}{2}\right)^n\sum_{k=0}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\sum_{i=0}^{k}\binom{n+1}{i}}{2^n+1}.$$

Although we have assumed that $p<1$ we can verify, that $F(n,1)=1/2$ as expected. We can not use $F(n,0)$ because $\Pr(n,0,k)$ is not a valid probability distribution over keys of length n.

3.2 Asymptotics

Based on Figure 1 we will now try to show that for all $p<1$ holds $\lim_{n\to\infty}F(n,p)=0$. This will be unwelcome result. It means that even if we encrypt nearly the entire plaintext up to some small fraction, this small fraction is still sufficient to reduce attack complexity to a negligible fraction compared to full text encryption.

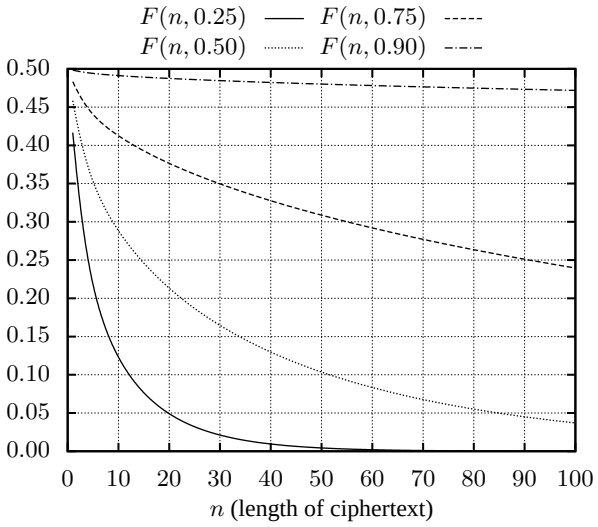

Fig. 1. This graph indicates that $\lim_{n\to\infty}F(n,p)=0$.

Since we want to prove that the limit goes to zero, it is possible to simplify the proof by realizing that $F(n,p)\geq 0$ and show that some simpler upper bound $f_0(n,p)+f_1(n,p)+f_2(n,p)\geq F(n,p)$ goes to zero too. We choose $f_i(n,p)$ as follows

$$f_0(n,p)=\frac{1}{2^{n+1}},$$

$$f_1(n,p)=\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n\sum_{k=0}^{\alpha}\left(\frac{p}{2-p}\right)^k\binom{n}{k}S_k^{n+1},$$

$$f_2(n,p)=\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n\sum_{k=\alpha}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}S_k^{n+1},$$

where S_k^{n+1} denotes $\sum_{i=0}^{k}\binom{n+1}{i}$ and α is $\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}$. To prove the main limit it is sufficient to show that for all $i \in \{0,1,2\}$ $\lim_{n\to\infty} f_i(n,p)$ equals to zero. For $i=0$ it is trivial so we move to $i=1$. In the proof we will use following lemma.

Lemma 1 (for proof see [5]). *Let $\varphi(n)$ be any function satisfying $\lim_{n\to\infty}\varphi(n)=\infty$. Then*

$$\lim_{n\to\infty}\frac{\sum_{k=0}^{n/2-\varphi(n)\sqrt{n}}\binom{n}{k}}{2^n}=0.$$

Theorem 3. *Let $p<1$. Then $\lim_{n\to\infty} f_1(n,p)=0$.*

Proof. Firstly we replace $f_1(n,p)$ with even simpler upper bound:

$$f_1(n,p)\le\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n S_\alpha^{n+1}\sum_{k=0}^{\alpha}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\le$$

$$\le\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n S_\alpha^{n+1}\sum_{k=0}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}=$$

$$=\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n S_\alpha^{n+1}\left(\frac{2}{2-p}\right)^n=\frac{S_\alpha^{n+1}}{2^{n+1}}$$

Now we can set $\varphi(n)=\frac{(n-1)^{\frac{5}{8}}+1}{2\sqrt{n}}$ and use lemma 1:

$$\lim_{n\to\infty}\frac{S_\alpha^{n+1}}{2^{n+1}}=\lim_{n\to\infty}\frac{\sum_{i=0}^{\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}}\binom{n+1}{i}}{2^{n+1}}=0.$$

Becase $f_1(n,p)\ge 0$ the theorem is proved. □

In the proof for $i=2$ we will utilize another two lemmas.

Lemma 2 (for proof see [6] equation 9.98). *Let $|k|\le\frac{1}{2}n^{\frac{5}{8}}$. Then binomial coefficient around center for $n\to\infty$ can be aproximated as follows:*

$$\binom{n}{\frac{n}{2}-k}=\frac{2^n}{\sqrt{\frac{\pi}{2}n}}e^{-2\frac{k^2}{n}}\left(1+O\left(n^{-\frac{1}{8}}\right)\right)$$

Lemma 3. *Let $a_k=\left(\frac{p}{2-p}\right)^k\binom{n}{k}$ for $k\in\{0,1,\dots,n\}$. Then the following inequality holds:*

$$\forall p\in(0,1)\;\exists m\;\forall n>m\;\forall k\ge\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}:a_k>a_{k+1}.$$

Proof. We rewrite theorem inequality as a fraction. So we get $\forall p\in(0,1)\;\exists m\;\forall n>m\;\forall k\ge\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}$:

$$\frac{a_{k+1}}{a_k}<1\Leftrightarrow\frac{p}{2-p}\frac{n-k}{k+1}<1\Leftrightarrow\frac{n-k}{k+1}<\frac{2-p}{p}.$$

Because $\frac{n-k}{k+1}<\frac{n-k}{k}$ it is sufficient to find an arbitrary m that $\forall n>m\;\forall k\ge\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}$:

$$\frac{n-k}{k}<\frac{2-p}{p}\Leftrightarrow n<k+k\frac{2-p}{p}\Leftrightarrow n<k\frac{2}{p}$$

By solving this inequality we get that it holds for every $k>\frac{p}{2}n$. Because we start with $k=\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}$, we will now solve for which n holds:

$$\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}>\frac{p}{2}n\Leftrightarrow 1-\frac{1}{n^{\frac{3}{8}}}>p\Leftrightarrow n>\left(\frac{1}{1-p}\right)^{\frac{8}{3}}.$$

Now we have showed that for all $p\in(0,1)$, if we set m to $\left(\frac{1}{1-p}\right)^{\frac{8}{3}}$, then

$$\forall n>m\;\forall k\ge\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}:a_k>a_{k+1}.$$

□

Now we are able to proof the last theorem on the limit of function f_2.

Theorem 4. *Let $p<1$. Then $\lim_{n\to\infty} f_2(n,p)=0$.*

Proof. Again we replace $f_2(n,p)$ with even simpler upper bound:

$$\frac{1}{2^{n+1}}\left(\frac{2-p}{2}\right)^n\sum_{k=\alpha}^{n}\left(\frac{p}{2-p}\right)^k\binom{n}{k}\underbrace{S_k^{n+1}}_{\le 2^{n+1}}\le$$

$$\le\left(\frac{2-p}{2}\right)^n\sum_{k=\alpha}^{n}\underbrace{\left(\frac{p}{2-p}\right)^k\binom{n}{k}}_{a_k}$$

Now we use lemma 3 and the fact that we are interested in limit for $n\to\infty$. Thus for large enough n we can upper bound the last equation by

$$\left(\frac{2-p}{2}\right)^n(n-\alpha+1)\,a_\alpha=$$

$$=\left(\frac{2-p}{2}\right)^n(n-\alpha+1)\left(\frac{p}{2-p}\right)^\alpha\binom{n}{\alpha}\le$$

$$\le\left(\frac{2-p}{2}\right)^n n\left(\frac{p}{2-p}\right)^\alpha\binom{n}{\alpha}=$$

$$=\left(\frac{2-p}{2}\right)^n\left(\frac{p}{2-p}\right)^{\frac{n}{2}}\left(\frac{2-p}{p}\right)^{\frac{1}{2}n^{\frac{5}{8}}}n\binom{n}{\alpha}=$$

$$=\frac{[(2-p)p]^{\frac{n}{2}}}{2^n}\left(\frac{2-p}{p}\right)^{\frac{1}{2}n^{\frac{5}{8}}}n\binom{n}{\frac{n}{2}-\frac{1}{2}n^{\frac{5}{8}}}.$$

In the sequel we get rid of binomial coefficient by applying lemma 2. We omit the $1+O(n^{-\frac{1}{8}})$ factor from following equations to save space.

$$\frac{[(2-p)p]^{\frac{n}{2}}}{2^n}\left(\frac{2-p}{p}\right)^{\frac{1}{2}n^{\frac{5}{8}}}n\frac{2^n}{\sqrt{\frac{\pi}{2}n}}e^{-2\frac{\left(\frac{1}{2}n^{\frac{5}{8}}\right)^2}{n}}=$$

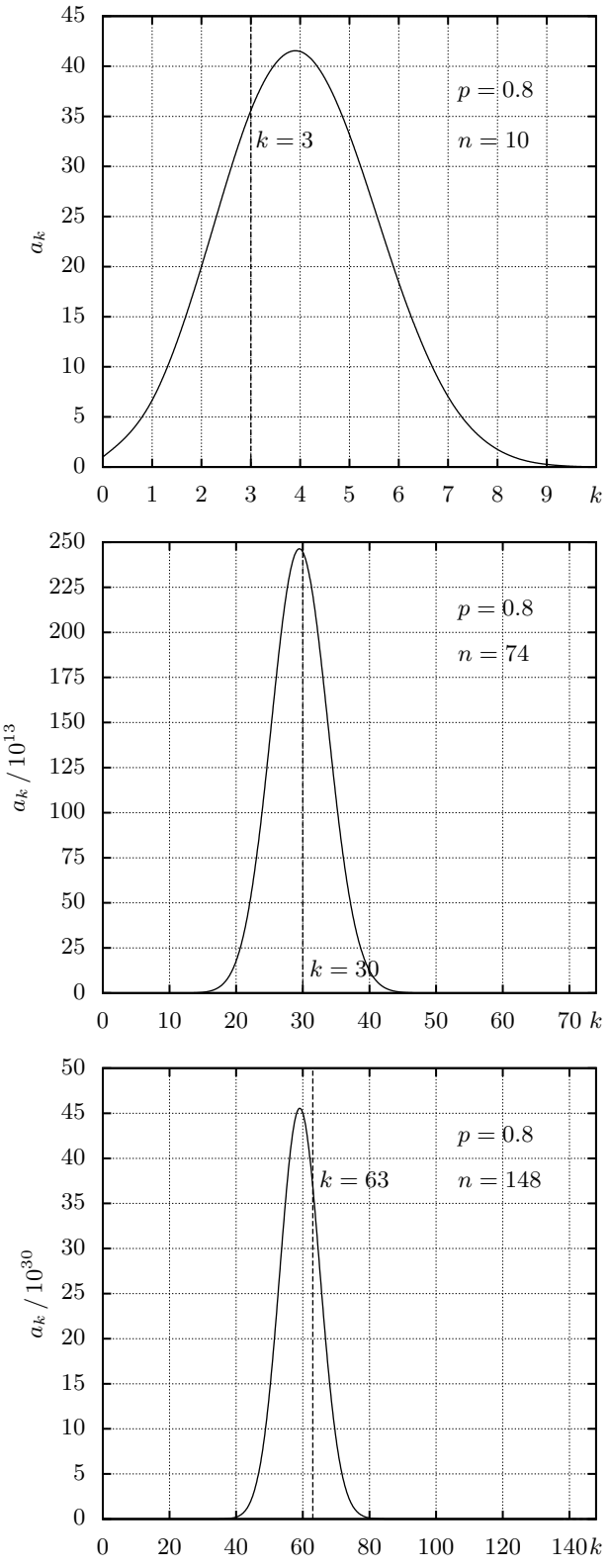

Fig. 2. These graphs illustrate how value of a_k starts to decrease from $k = \frac{n}{2} - \frac{1}{2}n^{\frac{5}{8}}$ if large enough n is used ($n > m$). For $p = 0.8$ based on lemma 3 we get that m approximately equals to 73.1.

$$= \sqrt{\frac{2}{\pi}}[(2-p)p]^{\frac{n}{2}} \left(\frac{2-p}{p}\right)^{\frac{1}{2}n^{\frac{5}{8}}} \sqrt{n}\, e^{-\frac{1}{2}\sqrt[4]{n}}$$

We want to prove now that the last equation goes to zero as n approaches ∞. We will do so by showing that logarithm of the equation goes to $-\infty$. We also omit constant factor $\sqrt{2/\pi}$ as it is irrelevant in this context.

$$\frac{n}{2}\ln(2-p)p + \frac{1}{2}n^{\frac{5}{8}}\ln\left(\frac{2-p}{p}\right) + \frac{1}{2}\ln n - \frac{1}{2}\sqrt[4]{n} + O\left(n^{-\frac{1}{8}}\right)$$

Since $\frac{n}{2}\ln(2-p)p$ is most influencing summand as n goes to infinity and $\ln(2-p)p < 0$ we have proved that the equation goes to $-\infty$. If we again omit the $1+O(n^{-\frac{1}{8}})$ factor from the right-hand side of inequality we get for large enough n that

$$0 \le f_2(n,p) \le \sqrt{\frac{2}{\pi}}[(2-p)p]^{\frac{n}{2}} \left(\frac{2-p}{p}\right)^{\frac{1}{2}n^{\frac{5}{8}}} \sqrt{n}\, e^{-\frac{1}{2}\sqrt[4]{n}}.$$

As we have proved that the right-hand side goes to zero as n goes to infinity, we are done. □

4 Conclusion

In this paper we have showed that even the coin flipping selector tremendously decreases the security of selective encryption for any $p < 1$. In other words it means that even if we encrypt nearly the entire plaintext up to some small fraction, this small fraction is still enough to reduce attack complexity to negligible fraction compared to full text encryption. The same result holds for random bit selector from [4] if the attacker and the attack complexity from this paper is assumed. As a conclusion we can say, that every studied selector significantly degrades security even if the encrypted fraction is closed to 1 for large enough messages.

References

1. Shi, C., Bhargava, B.: A fast mpeg video encryption algorithm. In: Proceedings of the sixth ACM international conference on Multimedia, ACM Press (1998) 81–88
2. Droogenbroeck, M.V., Benedett, R.: Techniques for a selective encryption of uncompressed and compressed images. In: Proceedings of ACIVS 2002 (Advanced Concepts for Intelligent Vision Systems), Ghent, Belgium (2002) 90 – 97
3. Liu, X., Eskicioglu, A.M.: Selective encryption of multimedia content in distribution networks: Challenges and new directions. ASTED International Conference on Communications, Internet and Information Technology (CIIT 2003) (2003)
4. Ostertág, R., Košinár, P.: Analýza selektorov pre selektívne šifrovanie. In: ITAT 2006: Information Technologies-Applications and Theory, Seňa: PONT (2006) 131–137

5. Olejár, D., Stanek, M.: On cryptographic properties of random Boolean functions. J.UCS: Journal of Universal Computer Science **4** (1998) 705–718
6. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1994)

On PCGS and FRR-automata

Dana Pardubská[⋆1], Martin Plátek[⋆⋆2], and Friedrich Otto[3]

[1] Department of Computer Science, Comenius University, Bratislava
pardubska@dcs.fmph.uniba.sk
[2] Department of Computer Science, Charles University, Prague
Martin.Platek@mff.cuni.cz
[3] Fachbereich Elektrotechnik/Informatik, Universität Kassel, Kassel
otto@theory.informatik.uni-kassel.de

Abstract. *This paper presents the second part of the technical report [7] in which the study of the relation between Parallel Communicating Grammar Systems (*PCGS*) and Freely Rewriting Restarting Automata (*FRR*) has been initiated. The first part of [7] is presented in [6]. Here, the distribution and generation complexity for* PCGS *are introduced and studied. It is shown that analysis by reduction for* PCGS *with distribution complexity bounded by a constant k and generation complexity bounded by some other constant j can be implemented by strongly linearized deterministic* FRR*-automata with k rewrites per cycle. We show infinite hierarchies of classes of languages based on the parameters k, j and on the notion of* skeleton*.*

1 Introduction

This paper deals with the comparison of Freely Rewriting Restarting Automata (FRR, [4]) and Parallel Communicating Grammar Systems (PCGS, [1, 8]). Namely, the so-called linearized FRR-automaton is used for this purpose. The motivation for our study is the usefulness of both models in computational linguistics.

Freely rewriting restarting automata form a suitable tool for modelling the so-called *analysis by reduction*. Analysis by reduction in general facilitates the development and testing of categories for syntactic and semantic disambiguation of sentences of natural languages. The Functional Generative Description for the Czech language developed in Prague (see, e.g., [2]) is based on this method.

FRR automata work on so-called *characteristic languages*, that is, on languages with auxiliary symbols (categories) included in addition to the input symbols. The proper language is obtained from a characteristic language by removing all auxiliary symbols from its sentences. By requiring that the automata considered are *linearized* we restrict the number of auxiliary symbols allowed on the tape by a function linear in the number of terminals on the tape. We mainly focus on deterministic restarting automata in order to ensure the *correctness preserving property* for the analysis, i.e., after any restart in an accepting computation the content of the tape is a word from the characteristic language. In fact, we mainly consider *strongly lexicalized* restarting automata. This additional restriction requires that all rewrite operations are deletions.

Parallel Communicating Grammar Systems are able to handle creations of copies of generated strings and their regular mappings in a natural way. This ability strongly resembles the generation of coordinations in Czech (and some other natural languages) sentences, where coordinations are certain contiguous segments (not only lexicalized elements). However, the synonymy of coordinations has not yet been modelled appropriately.

In this paper the notions of distribution and generation complexity for PCGS are introduced and studied. It is shown that analysis by reduction for PCGS with distribution complexity bounded by a constant k and generation complexity bounded by some other constant j can be implemented by strongly linearized deterministic FRR-automata with k rewrites per cycle. We show infinite hierarchies of classes of languages based on the parameters k, j and on the notion of *skeleton*. The notion of skeleton is introduced in order to model the principle of so-called segments in (Czech) sentences (or in text). The elements of skeletons are so-called *islands*, which serve to model the so-called separators of segments (see [3]).

2 Basic notions

A *freely rewriting restarting automaton*, abbreviated as FRR-automaton, is described by an 8-tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$. It consists of a finite-state control, a flexible tape, and a read/write window of a fixed size $k \geq 1$. Here Q denotes a finite set of (internal) states that contains the initial state q_0, Σ is a finite input alphabet, and Γ is a finite tape alphabet that contains Σ. The elements of $\Gamma \setminus \Sigma$ are called *auxiliary symbols*. The additional symbols $¢, \$ \notin \Gamma$ are used as markers for the left and right end of the workspace, respectively. They cannot be removed from the tape. The behavior of M is described by a transition function δ that associates transition steps to certain

⋆ Partially supported by the Slovak Grant Agency for Science (VEGA) under contract "Theory of Models, Complexity and Algorithms".

⋆⋆ Partially supported by the Grant Agency of the Czech Republic under Grant-No. 405/08/0681 and by the program Information Society under project 1ET100300517.

pairs of the form (q, u) consisting of a state q and a possible content u of the read/write window. There are four types of transition steps: *move-right steps*, *rewrite steps*, *restart steps*, and *accept steps*. A *move-right step* simply shifts the read/write window one position to the right and changes the internal state. A *rewrite step* causes M to replace a non-empty prefix u of the content of the read/write window by a shorter word v, thereby shortening the length of the tape, and to change the state. Further, the read/write window is placed immediately to the right of the string v. A *restart step* causes M to place its read/write window over the left end of the tape, so that the first symbol it sees is the left sentinel ¢, and to reenter the initial state q_0. Finally, an *accept step* simply causes M to halt and accept.

A *configuration* of M is described by a string $\alpha q \beta$, where $q \in Q$, and either $\alpha = \varepsilon$ (the empty word) and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$; here q represents the current state, $\alpha\beta$ is the current content of the tape, and it is understood that the window contains the first k symbols of β or all of β when $|\beta| \leq k$. A *restarting configuration* is of the form $q_0 ¢ w \$$, where $w \in \Gamma^*$.

Any computation of M consists of certain phases. A phase, called a *cycle*, starts in a restarting configuration. The window is shifted along the tape by move-right and rewrite operations until a restart operation is performed and thus a new restarting configuration is reached. If no further restart operation is performed, the computation necessarily finishes in a halting configuration – such a phase is called a *tail*. It is required that in each cycle M performs at least one rewrite step. As each rewrite step shortens the tape, we see that each cycle reduces the length of the tape. We use the notation $u \vdash_M^c v$ to denote a cycle of M that begins with the restarting configuration $q_0 ¢ u \$$ and ends with the restarting configuration $q_0 ¢ v \$$; the relation $\vdash_M^{c^*}$ is the reflexive and transitive closure of $\vdash_M^c$.

A word $w \in \Gamma^*$ is *accepted* by M, if there is a computation which starts from the restarting configuration $q_0 ¢ w \$$, and ends with an application of an accept step. By $L_C(M)$ we denote the language consisting of all words accepted by M. It is the *characteristic language* of M.

By Pr^Σ we denote the projection from Γ^* onto Σ^*, that is, Pr^Σ is the morphism defined by $a \mapsto a$ ($a \in \Sigma$) and $A \mapsto \varepsilon$ ($A \in \Gamma \setminus \Sigma$). If $v := \mathsf{Pr}^\Sigma(w)$, then v is the *Σ-projection* of w, and w is an *expanded version* of v. For a language $L \subseteq \Gamma^*$, $\mathsf{Pr}^\Sigma(L) := \{ \mathsf{Pr}^\Sigma(w) \mid w \in L \}$.

In recent papers restarting automata were mainly used as acceptors. The (*input*) *language* accepted by a restarting automaton M is the set $L(M) := L_C(M) \cap \Sigma^*$. Here, motivated by linguistic considerations to model the analysis by reduction with parallel processing, we are rather interested in the so-called *proper language of* M, which is the set of words $L_P(M) := \mathsf{Pr}^\Sigma(L_C(M))$. Hence, a word $v \in \Sigma^*$ belongs to $L_P(M)$ if and only if there exists an expanded version u of v such that $u \in L_C(M)$.

For each type X of restarting automata, we use $\mathcal{L}_C(\mathsf{X})$ and $\mathcal{L}_P(\mathsf{X})$ to denote the class of all characteristic languages and the class of all proper languages of automata of this type.

Following basic properties of FRR-automata are often used in proofs.

(Correctness Preserving Property.) Each deterministic FRR-automaton M is *correctness preserving*, i.e., if $u \in L_C(M)$ and $u \vdash_M^{c^*} v$, then $v \in L_C(M)$, too.

(Cycle Pumping Lemma.) For any FRR-automaton M, there exists a constant p such that the following property holds. Assume that $uxvyz \vdash_M^c ux'vy'z$ is a cycle of M, where $u = u_1 u_2 \cdots u_n$ for some non-empty words $u_1, \ldots, u_n$ and an integer $n > p$. Then there exist $r, s \in \mathbb{N}_+$, $1 \leq r < s \leq n$, such that
$u_1 \cdots u_{r-1}(u_r \cdots u_{s-1})^i u_s \cdots u_n xvyz \vdash_M^c$
$u_1 \cdots u_{r-1}(u_r \cdots u_{s-1})^i u_s \cdots u_n x'vy'z$
holds for all $i \geq 0$, that is, $u_r \cdots u_{s-1}$ is a "pumping factor" in the above cycle. Similarly, such a pumping factor can be found in any factorization of length greater than p of v or z as well as in any factorization of length greater than p of a word accepted in a tail computation.

We focus our attention on FRR-automata, for which the use of auxiliary symbols is less restricted than in [4].

Definition 1. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an* FRR*-automaton, $|x|_K$ denotes the number of occurrences of symbols from K in word x.*

(a) *The* FRR*-automaton M is called* linearized *if there exists a constant $j \in \mathbb{N}_+$ such that $|w|_{\Gamma-\Sigma} \leq j \cdot |w|_\Sigma + j$ for each $w \in L_C(M)$.*
(b) *M is called* strongly linearized *if it is linearized, and if each of its rewrite operations just deletes some symbols.*

Since linearized FRR automata use linear space only, we have the following:

Corollary 1. *If M is a linearized* FRR*-automaton, then the proper language $L_P(M)$ is context-sensitive.*

In what follows we are mainly interested in strongly linearized FRR-automata and their proper languages. We denote by (S)LnRR the class of (strongly) linearized deterministic FRR-automata, by N(S)LnRR the class of nondeterministic (strongly) linearized FRR-automata, and by t-$\mathcal{A}$ the subclass of $\mathcal{A}$-automata which execute at most t rewrite steps in any cycle.

2.1 Parallel Communicating Grammar Systems

A PCGS *of degree* m, $m \geq 1$, is an $(m+1)$-tuple $\Pi = (G_1, \ldots, G_m, K)$, where for all $i \in \{1, \ldots, m\}$, $G_i = (N_i, T, S_i, P_i)$, so-called component grammars, are regular grammars satisfying $N_i \cap T = \emptyset$ and $K \subseteq$

$\{Q_1,\ldots,Q_m\}\bigcap\bigcup_{i=1}^{m} N_i$ is a set of special symbols, called *communication symbols*.

A *configuration* is an m-tuple $C=(x_1,\ldots,x_m)$, $x_i=\alpha_i A_i$, $\alpha_i\in T^*, A_i\in(N_i\cup\varepsilon)$; we call x_i the *i-th component of the configuration* (resp. component). The *nonterminal cut* of configuration C is the $m-$tuple $N(C)=(A_1,A_2,\ldots,A_m)$. If the nonterminal cut $N(C)$ contains at least one communication symbol, it is denoted $NC(C)$ and called an NC-cut.

We say that a *configuration* $X=(x_1,\ldots,x_m)$ *directly derives a configuration* $Y=(y_1,\ldots,y_m)$, and write $X\Rightarrow Y$, if Y is derived from X by one *generative* or *communication* step (see below). Informally, in a communication step any occurrence of a communication symbol Q_i in X is substituted by the i-th component of X (assuming that this component does not contain any communication symbol).

1. (Generative step) If $|x_i|_K=0$ for all i , $1\le i\le m$, then
   $x_i\overset{G_i}{\Rightarrow}y_i$ for $x_i\in T^*N_i$ and
   $y_i=x_i$ for $x_i\in T^+$.
2. (Communication step) If $|x_i|_K>0$ for some i, $1\le i\le m$, then for each k such that $x_k=z_kQ_{j_k}$, $z_k\in T^*, Q_{j_k}\in K$, the following is true:
   (a) if $|x_{j_k}|_K=0$, then $y_k=z_kx_{j_k}$ and $y_{j_k}=S_{j_k}$;
   (b) if $|x_{j_k}|_K=1$, then $y_k=x_k$.

For all remaining indices t, for which x_t does not contain a communication symbol and Q_t has not occurred in any of the x_i's, we put $y_t=x_t$.

Now, we describe the derivations in PCGSs. A *derivation* of a PCGS Π is a sequence of configurations $D=C_1,C_2,\ldots,C_t$, where $C_i\Rightarrow C_{i+1}$ in Π. If the first component of C_t is a terminal word w, then we usually write $D(w)$ instead of D. Analogously, we denote by $W(D)$ the terminal word generated within the derivation D. Every derivation can be viewed as a sequence of *generative* and *communication steps*, too.

If no communication symbol appears in any of the component grammars, then we perform a *generative step* consisting of rewriting steps synchronously performed in each of the component grammars $G_i, 1\le i\le m$. If any of the components is a terminal string, it is left unchanged. If any of the component grammars contains a nonterminal that cannot be rewritten, the derivation is blocked. If the first grammar G_1 contains a terminal word w, the derivation finishes and w is the word generated by Π in this derivation.

If a communication symbol is present in any of the components, then a *communication step* is performed. It consists of replacing those communication symbols with the phrases they refer to for which the phrases do not contain communication symbols. Such an individual replacement is called a *communication*. Obviously, in one communication step at most $m-1$ communications can be performed. If some communication symbol was not replaced in this communication step, it may be replaced in one of the next communication steps. Communication steps are performed until no more communication symbols are present or the derivation is blocked, because no communication symbol can be replaced in the last communication step.

The *(terminal) language* $L(\Pi)$ generated by a PCGS Π is a set of the terminal words generated by G_1 (in cooperation with the other grammars):

$$L(\Pi)=\{\,\alpha\in T^*\mid(S_1,\ldots,S_m)\Rightarrow^+(\alpha,\beta_2,\ldots,\beta_m)\,\}.$$

Let $D=D(w)=C_0,C_1,\ldots,C_t$ be a derivation of w by Π; $D(w)$, Π and w are fixed in what follows. With derivation $D(w)$, several notions can be associated which help to analyze the derivation of Π and to unambiguously determine w.

The *trace* of a (sub)derivation D is the sequence $T(D)=N(C_0)N(C_1)\ldots N(C_t)$ of the nonterminal cuts of the individual configurations of D.

The *NC-sequence* is defined analogously; $NCS(D)$ is the sequence of the NC-cuts of the configurations in the (sub)derivation D. Let us recall that any NC-cut contains at least one communication symbol.

A *cycle in a derivation* is a subsequence $N(C),N(C_1),\ldots,N(C_j),N(C)$ of nonterminal cuts of the derivation[4] in which the first and the last cuts ($N(C)$) are the same. If $N(C)$ is an NC-cut, and none of the intermediate cuts $N(C_i)$ is an NC-cut, then the cycle is called a *communication cycle*. A *generative cycle* is defined analogously, we only require that *none* of its cuts is an NC-cut.

Note that, if there is a cycle in the derivation $D(w)$, then manifold repetition of the cycle is possible and the resulting derivation is again a derivation of some terminal word. We call a derivation $D(w)$ *reduced*, if each repetition of each of its cycles leads to a *longer* terminal word ω; $|w|<|\omega|$. Obviously, to every derivation $D(w)$ there is an equivalent reduced derivation $D'(w)$.

A *generative section* is a non-empty sequence of generative steps between two consecutive communication steps in $D(w)$[5], resp. before the first and/or after the last communication steps in $D(w)$.

The *degree of generation* $DG(D(w))$ is the number of generative sections of $D(w)$. In the following we consider only PCGS **without communication cycles**, i.e., $DG(D(w))$ is bounded by a constant depending only on Π.

$\mathbf{g(i,j)}$ ($\mathbf{g(i,j,D(w))}$) denotes the terminal part generated by G_i within the j-th generative section of $D(w)$, we call it the (i,j)-(generative) factor (of $D(w)$);

$\mathbf{n(i,j)}$ ($\mathbf{n(i,j,D(w))}$) denotes the number of occurrences of $g(i,j)$ in w;

[4] More precisely it is a subsequence of trace of the derivation.

[5] Note that if some communication cut contains more than one communication symbol, then there might be no generative step between two communication steps.

$\mathbf{g(i,j,l)}$ denotes the l-th occurrence of $g(i,j)$ in w, we call it the (i,j,l)-(generative) factor.

The *communication structure* $CS(D(w))$ of $D(w)$ is
$CS(D(w)) = (i_1,j_1,l_1),(i_2,j_2,l_2),\ldots,(i_r,j_r,l_r)$,
where $w = g(i_1,j_1,l_1),g(i_2,j_2,l_2)\ldots g(i_r,j_r,l_r)$. The *set* of these indices is denoted $I(D(w))$.

$\mathbf{N(j,D(w))} = \Sigma_i\, n(i,j,D(w))$, where the sum is taken over such i for which $\exists s : i = i_s \;\&\; (i_s,j_s,l_s) \in I(D(w))$.

The *degree of distribution* $DD(D(w))$ of $D(w)$ is the maximum over all (defined) $N(j,D(w))$.

Now, we are ready to introduce the notions of *distribution complexity* and *generation complexity*. First, the distribution complexity of a derivation D (denoted $DD(D)$) is the degree of distribution introduced above.

Then, the distribution complexity of a language and the associated complexity class are defined in the usual way (always considering the corresponding maximum): distribution complexity of a derivation $\rightsquigarrow$ distribution complexity of a word $\rightsquigarrow$ distribution complexity of a language L (denoted $DD(L)$) as a function of the length of the word $\rightsquigarrow$ $f(n) - DD$ as class of languages whose distribution complexity is bounded by $f(n)$.

The generation complexity is introduced analogously. Here, we are mainly interested in the classes of languages with t-DD and/or with j-DG for some natural numbers j,t. We denote by j-t-DDG the class of languages such that, to any language L of this class, there is a PCGS Π such that $L(\Pi) = L$, and $DD(L(\Pi)) = t$, $DG(L(\Pi)) = j$.

Relevant observations about derivations of PCGS (see [5] for more information) are summarized in the following facts:

Fact 1 *Let Π be a* PCGS *without a communication cycle. Then there are constant $d(\Pi),\ell(\Pi),s(\Pi)$ such that*

1. *the number $n(i,j)$ of occurrences of individual $g(i,j)'s$ in a reduced derivation $D(w)$ is bounded by $d(\Pi); n(i,j) \leq d(\Pi)$;*
2. *the length of the communication structure for every reduced derivation $D(w)$ is bounded by $\ell(\Pi)$;*
3. *the cardinality of the set of possible communication structures corresponding to reduced derivations by Π is bounded by $s(\Pi)$.*

Fact 2 *Let Π be a* PCGS *without a communication cycle, $D(w)$ a reduced derivation of a terminal word w. Then there is a constant $e(\Pi)$ such that, if more than $e(\Pi)$ generative steps of one generative section are performed, then at least one $g(i,j,D(w))$ is changed (see Example 1 in [7]).*

3 Bounded degree of distribution

We start the section showing that a language generated by a PCGS Π with constant distribution complexity can be analyzed (by reduction) by a t-SLnRR-automaton M.

In fact, the result follows from the analysis of the proof of Theorem 1 ([7]). For better understanding and to motivate the notions defined below we sketch the mentioned proof (from [7]).

The high-level idea is to merge the terminal word w with the information describing its reduced derivation $D(w)$ in a way allowing simultaneously the "simulation/reduction" of the derivation $D(w)$ and the correctness checking. Analysis by reduction is based on the deletion of the parts of a (characteristic) word which correspond to parts generated within one generative cycle. We call such a merged (characteristic) word Π-description of w.
Let $(\alpha_1 A_1,\ldots,\alpha_m A_m)$ be the configuration at the beginning of the j-th generative section,
$(A_1,\ldots,A_m),(\alpha_{1,1}A_{1,1},\ldots,\alpha_{1,m}A_{1,m}),\ldots$
$(\alpha_{1,1}\alpha_{2,1}\ldots\alpha_{s,1}A_{s,1},\ldots,\alpha_{1,m}\alpha_{2,m}\ldots\alpha_{s,m}A_{s,m})$
the sub-derivation corresponding to this generative section. Merging the description of this sub-derivation into $g(i,j,l)$ we obtain the extended version of $g(i,j,l)$:

$$[b,i,j,l]\begin{pmatrix}A_1\\A_2\\\cdots\\A_m\end{pmatrix}\alpha_{1,i}\begin{pmatrix}A_{1,1}\\A_{1,2}\\\cdots\\A_{1,m}\end{pmatrix}\alpha_{2,i}\begin{pmatrix}A_{2,1}\\A_{2,2}\\\cdots\\A_{2,m}\end{pmatrix}\ldots$$
$$\ldots\alpha_{s,i}\begin{pmatrix}A_{s,1}\\A_{s,2}\\\cdots\\A_{s,m}\end{pmatrix}[e,i,j,l].$$

Such a description of $g(i,j,l)$ is denoted ex-$g(i,j,l)$. We use ex-$g(i,j,l)$ to merge the (topological) information about derivation $D(w)$ into w. Obviously, we can speak about *traces* and *factor cycles* in ex-$g(i,j,l)$ similarly as we speak about traces and generative cycles in derivations.

Let $w, D(w), ex$-$g(i,j,l)$, be as above. Replace any $g(i,j,l)$ in w by ex-$g(i,j,l)$; the result is denoted ex-w. Then, concatenating the NC-sequence of $D(w)$, the communication structure given by $D(w)$, and ex-w we obtain the *Π-description of w*:

$$\Pi d(D(w)) = NCS(D(w))CS(D(w))ex\text{-}w.$$

Observations. Let $\Pi d(D(w))$ be the Π-description of w.

(a) When a reduced derivation $D(w)$ is taken, then the length of $\Pi d(D(w))$ is bounded from above by $c_\Pi \cdot |w| + c_\Pi$, where c_Π is a constant depending on Π only.
(b) From $\Pi d(D(w))$ the terminal word w is easily obtained by deleting all symbols which are not terminal symbols of Π.
(c) Let $T(D(w))$ be the trace of $D(w)$, and $T(\Pi) := \{T(D(w)) \mid w \in L(\Pi)\}$. Then, $T(\Pi)$ is a regular language, and the sets of NC-cuts and communication sequences of Π are finite. Note that a finite automaton is also able to check whether a given string x is a correct ex-$g(i,j,l)$, $NCS(D(w))$, or $CS(D(w))$ given by Π.

Analyzing the proof of Theorem 1 from [7] we have the following consequence. The construction of a k-SLnRR-automaton M accepting the characteristic language $L_C(M) = \{\Pi d(D(w)) \mid w \in L(\Pi)\}$ is outlined in [7].

Corollary 2. *For all $k \in \mathbb{N}$,* k-DD $\subseteq \mathcal{L}_{\rm P}(k$-SLnRR$)$.

For $t \in \mathbb{N}_+$, separation of PCGSs of distribution complexity t from the proper languages of nondeterministic linearized FRR-automata with at most $t-1$ rewrites per cycle, is done with the help of the language

$$L_t := \{\, c_1 w d \cdots c_t w d \mid w \in \{a,b\}^* \,\},$$

where $\Sigma_1 := \{c_1, \ldots, c_t, d\}$ is a new alphabet disjoint from $\Sigma_0 := \{a, b\}$.

Proposition 1. *For all $t \in \mathbb{N}_+$,*
$L_t \in \mathcal{L}(t$-DD$) \setminus \mathcal{L}_{\rm P}((t-1)$-NLnRR$)$.

Proof. It is not hard to show that $L_t \in \mathcal{L}(t$-DD$)$. We use a PCGS with $t+1$ component grammars for that:
$(S_1, S_2, \ldots, S_{t+1}) \Rightarrow^*$
$\Rightarrow^* (c_1 Q_2, c_2 Q_3, \ldots, c_t Q_{t+1}, wd)$
$\Rightarrow^* (c_1 w d c_2 w d \ldots c_t w d, S_2, \ldots, S_t, S_{t+1})$.

For the lower-bound part we use a similar technique as in [4].

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a nondeterministic linearized FRR-automaton that executes at most $t-1$ rewrites per cycle, where $\Sigma := \Sigma_0 \cup \Sigma_1$. Assume that $L_{\rm P}(M) = L_t$ holds. Consider the word $w := c_1 a^n b^n d \cdots c_t a^n b^n d \in L_t$, where n is a large integer. Then there exists an expanded version $W \in \Gamma^*$ of w such that $W \in L_{\rm C}(M)$. Let W be a shortest expanded version of w in $L_{\rm C}(M)$. Consider an accepting computation of M on input W. Clearly this cannot just be an accepting tail, and hence, it begins with a cycle of the form $W \vdash_M^c W_1$. From the Correctness Preserving Property it follows that $W_1 \in L_{\rm C}(M)$, which implies that $w_1 := \mathsf{Pr}^{\Sigma}(W_1) \in L_{\rm t}$. As $|W_1| < |W|$, we see from our choice of W that $w_1 \neq w$, that is, $w_1 = c_1 x_1 d \cdots c_t x_1 d$ for some word $x_1 \in \Sigma_0^*$ of length $|x_1| < 2n$. However, in the above cycle M executes at most $t-1$ rewrite steps, that is, it cannot possibly rewrite each of the t occurrences of $a^n b^n$ into the same word x_1. It follows that $w_1 \notin L_t$, implying that $L_t \notin \mathcal{L}_{\rm P}((t-1)$-NLnRR$)$. □

As $\mathcal{L}(t$-DD$) \subseteq \mathcal{L}_{\rm P}(t$-SLnRR$)$, we obtain the following hierarchies from Proposition 1, where $\mathcal{L}_{\rm P}(t$-DD$)$ just denotes the class $\mathcal{L}(t$-DD$)$.

Theorem 1. *For all* X $\in$ {DD, LnRR, SLnRR, NLnRR, NSLnRR}*, and all $t \geq 1$,*

$$\mathcal{L}_{\rm P}(t\text{-X}) \subset \mathcal{L}_{\rm P}((t+1)\text{-X}) \subset \bigcup_{t \geq 1} \mathcal{L}_{\rm P}(t\text{-X}) \subset \mathcal{L}_{\rm P}(\text{X}).$$

4 Skeletons

In this part we define the notions of *skeleton* and *islands* whose introduction has been motivated by our attempt to model two basic kinds of coordinated segments in (Czech, German, Slovak) sentences. The islands in a level of skeleton serve to denote places of coordinated segments which are coordinated in a mutually dependent (bound) way. The different levels of islands serve for modelling the independence of segments. A technical example how to construct skeletons is given by the construction in the proof of [7] Theorem 1. In fact, skeletons are only defined for t-SLnRR-automata that fulfill certain additional requirements.

Definition 2. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a t-SLnRR-automaton for some $t \in \mathbb{N}_+$, and let $s \in \mathbb{N}_+$. Let $SK(s) = \{\, c_{i,j} \mid 1 \leq i \leq t, 1 \leq j \leq s \,\}$ be a subalphabet of cardinality $t \cdot s$ of $\Gamma' = \Gamma \cup \{¢, \$\}$. For each $j \in \{1, \ldots, s\}$, let $SK(s,j) = \{c_{1,j}, \ldots, c_{t,j}\}$ be the j-th* level *of $SK(s)$. We say that $SK(s)$ is an* s-skeleton (skeleton) *of M if the following holds:*

1. *For all $w \in L_{\rm C}(M)$ and all $c \in SK(s)$, $|w|_c \leq 1$, that is, w contains at most one occurrence of c.*
2. *Each rewrite operation of M deletes a single continuous factor from the actual contents of the window, and at that point the window must contain exactly one occurrence of a symbol from $SK(s)$. This symbol is either in the first or in the last position of the window.*
3. *If a cycle C of M contains a rewrite operation during which a symbol $c_{i,j} \in S(s,j)$ is in the first or last position of the window, then every rewrite operation during C is executed with some element of $S(s,j)$ in the first or last position of the window.*
4. *If $w \in L_{\rm C}(M)$, $w = xyz$, such that $|y| > k$, and y does not contain any element of $SK(s)$, then starting from the restarting configuration corresponding to w, M will execute at least one cycle before it accepts.*

The elements of $SK(s)$ are called islands *of M. We say that $SK(s)$ is a* left skeleton *of M, if M executes rewrite operations only with an island in the leftmost position of its window.*

Thus, in each cycle M performs up to t rewrite (that is, delete) operations, and during each of these operations a different island $c_{i,j}$ of the same level $SK(j)$ is inside the window. As there are s such levels, we see that there are essentially just s different ways to perform the rewrite steps of a cycle.

By $\mathcal{L}_{\rm P}(t$-SK$(s))$ (resp. by $\mathcal{L}_{\rm P}(t$-LSK$(s))$) we denote the class of proper languages of t-SLnRR-automata with s-skeletons (resp. with left s-skeletons). The corresponding classes of characteristic languages are denoted by $\mathcal{L}_{\rm C}(t$-SK$(s))$ (resp. by $\mathcal{L}_{\rm C}(t$-LSK$(s))$).

Observe that the symbols of the form $[b, i, s, l]$ in the construction of an s-SLnRR-automaton M accepting the language $L_C(M) = \{ \Pi d(D(w)) \mid w \in L(\Pi) \}$ play the role of islands for M, and their complete set is a left skeleton for M. This observation serves as the basis for the proof of the next corollary. Recall that s-t-DDG denotes the class of PCGSs that have simultaneously generation degree s and distribution degree t.

Corollary 3.
For all $s, t \in \mathbb{N}_+$, $\mathcal{L}(s\text{-}t\text{-DDG}) \subseteq \mathcal{L}_{\mathrm{P}}(t\text{-LSK}(s))$.

To separate PCGSs of generation complexity t and distribution complexity s from the class of proper languages of $(t-1)$-LSK(s)-automata we define language $L_{(t,s)}$. This language is based on a kind of bounded concatenation of L_t. For $s, t \in \mathbb{N}_+$ and $i \leq s$, let
$L_{(t)} := \{ c_1 w d \cdots c_t w d \mid w \in \{a, b\}^*, c_i \in \Sigma_i, d \in \Delta \}$, where $\Sigma_1, \ldots \Sigma_s, \Delta$ are new alphabets with empty intersection with $\{a, b\}$. Then,

$$L_{(t,s)} := (L_{(t)})^s.$$

Proposition 2. *For all* $s, t \in \mathbb{N}_+$,

(a) $L_{(t,s)} \in \mathcal{L}(s\text{-}t\text{-DDG})$,
(b) $L_{(t,s)} \notin \mathcal{L}_{\mathrm{P}}(t\text{-SK}(s-1))$ *for* $s > 1$, *and*
(c) $L_{(t,s)} \notin \mathcal{L}_{\mathrm{P}}((t-1)\text{-SK}(s))$ *for* $t > 1$.

Sketch of the proof. Note that $L_t = L_{(t)} = L_{(t,1)}$ when $|\Sigma_1| = \cdots = |\Sigma_t| = |\Delta| = 1$.
(a) For the upper-bound part we use a PCGS with $(t+s)$ component grammars, which realize s phases corresponding to s generative sections. The group of grammars $G_{s+1}, \ldots, G_{s+t}$ plays the role of $G_2, \ldots, G_{t+1}$ from the proof of Proposition 1, while the component grammars $G_1, \ldots, G_s$ play the role of grammar G_1 from that proof. At the end of the p-th generative section, there is a word $\omega_p i$ present in component grammar G_{s+1}, where $\omega_p = c_{1,p} w_p d_p \ldots c_{t,p} w_p d_p$ is a terminal word and i, $1 \leq i \leq s$, is a nonterminal symbol indicating that G_i is the grammar into which ω_p should be communicated. Finally, the synchronized communication concatenates all ω's in an appropriate[6] way in component grammar G_1.

(b) Assume that M is a t-SK$(s-1)$-automaton such that $L_{\mathrm{P}}(M) = L_{(t,s)}$. Thus, M has a $(s-1)$-skeleton $SK(s-1) = \{ c_{i,j} \mid 1 \leq i \leq t, 1 \leq j \leq s-1 \}$. Now assume that, for $i = 1, \ldots, s$, $w_i \in L_{t,i}$, that is, $w := w_1 w_2 \cdots w_s \in L_{(t,s)}$. Further, let W be an expanded version of w. For each cycle of M in an accepting computation on input W, there exists an index $j \in \{1, \ldots, s-1\}$ such that each rewrite step of this cycle is executed with an island $c_{i,j}$ in the left- or rightmost position of the window. From the proof of Proposition 1 we see that, for each of the factors $L_{t,j}$, t rewrite steps per cycle are required. Thus, each of the factors W_i must contain t islands, that is, W must contain at least $t \cdot s$ islands. However, as the word $W \in L_{\mathrm{C}}(M)$ contains at most a single occurrence of each symbol of the set $SK(s-1)$, and as $|SK(s-1)| = t \cdot (s-1)$, W can contain at most $t \cdot (s-1)$ islands. This contradicts the observation above, implying that $L_{(t,s)}$ is not the proper language of any t-SK$(s-1)$-automaton.
(c) For the lower-bound part recall Proposition 1 where $L_t \notin \mathcal{L}_{\mathrm{P}}((t-1)\text{-NLnRR})$ is shown to hold. From the proof it follows that $L_{(t,s)} \notin \mathcal{L}_{\mathrm{P}}((t-1)\text{-NLnRR})$. As $(t-1)$-SK(s)-automata are a special type of $(t-1)$-SLnRR-automata, the non-inclusion result in (c) follows. □

Next we consider the language $L_{pe} := \{ wcw^R \mid w \in \{0,1\}^* \}$. By taking the symbol c as an island, we easily obtain the following result.

Proposition 3. $L_{pe} \in \mathcal{L}_{\mathrm{P}}(2\text{-SK}(1))$.

On the other hand, this language cannot be accepted if we restrict our attention to left skeletons.

Proposition 4. $\forall s, t \in \mathbb{N}_+ : L_{pe} \notin \mathcal{L}_{\mathrm{P}}(t\text{-LSK}(s))$.

Proof. Assume that M is a t-LSK(s)-automaton such that $L_{\mathrm{P}}(M) = L_{pe}$, that is, M has a left skeleton $SK(s) = \{ c_{i,j} \mid 1 \leq i \leq t, 1 \leq j \leq s \}$. Let $w = (a^n b^n)^m$, where $n, m \in \mathbb{N}_+$ are sufficiently large, and let $z = wcw^R \in L_{pe}$. Then there exists a (shortest) expanded version $Z \in \Gamma^+$ of z such that $Z \in L_{\mathrm{C}}(M)$. Hence, the computation of M on input Z is accepting, but because of the Pumping Lemma it cannot just consist of an accepting tail, that is, it begins with a cycle $Z \vdash_M^{\mathrm{c}} V$, where $V \in L_{\mathrm{C}}(M)$ and $|V| < |Z|$. Thus, $v = \mathsf{Pr}^{\Sigma}(V) \in L_{pe}$, but $v \neq z$. In this cycle M performs up to t delete operations that each delete a continuous factor of Z to the right of an island $c_{i,j}$ for some level $j \in \{1, \ldots, s\}$. It follows that $v = w_1 c w_1^R$ for some word $w_1 \in \{a, b\}^*$ satisfying $|w_1| < |w|$, and that w_1 is obtained from w by deleting some factors, and w_1^R is obtained from w^R by deleting the corresponding reverse factors. When deleting a factor x within the prefix w to the right of an island $c_{i,j}$, then this means that this island "moves" to the right inside w, that is, from $c_{i,j}xy$ the factor $c_{i,j}y$ is obtained. Here we just consider the projection of Z onto $(SK(s,j) \cup \{a, b\})^*$. Now when the corresponding factor x^R is deleted from w^R, then it is to the right of an island $c_{i',j}$, that is, from $y^R c_{i',j} x^R$ the factor $y^R c_{i',j}$ is obtained. Thus, while for deleting the factor y of w the same island $c_{i,j}$ could be used in a later cycle, an island different from $c_{i',j}$ is needed for y^R. The same argument applies to the case that the roles of w and w^R are interchanged. This means that in the process of synchronously processing w

[6] The construction of PCGS heavily utilizes nondeterminism. In case of "wrong" nondeterministic choices the derivation is blocked.

and w^R, the same island can be used repeatedly in subsequence cycles within one of the two parts, but the corresponding deletions in the other part require new islands in each cycle. If w is of sufficient length, then it follows that $t \cdot s$ islands will not suffice. Hence, $L_{pe} \notin \mathcal{L}_{\mathrm{P}}(t\text{-LSK}(s))$. □

The results above yield the following consequences.

Theorem 2. *For all* $\mathsf{X} \in \{\mathsf{LSK}, \mathsf{SK}\}$, *and all* $s, t \geq 1$, *we have the following* proper *inclusions:*

(a) s-t-DDG $\subset (s+1)$-t-DDG.
(b) s-t-DDG $\subset s$-$(t+1)$-DDG.
(c) $\mathcal{L}_{\mathrm{P}}(t\text{-X}(s)) \subset \mathcal{L}_{\mathrm{P}}((t+1)\text{-X}(s))$.
(d) $\mathcal{L}_{\mathrm{P}}(t\text{-X}(s)) \subset \mathcal{L}_{\mathrm{P}}(t\text{-X}(s+1))$.
(e) s-t-DDG $\subseteq \mathcal{L}_{\mathrm{P}}(t\text{-LSK}(s)) \subseteq \mathcal{L}_{\mathrm{P}}(t\text{-SK}(s))$.
(f) $\mathcal{L}_{\mathrm{P}}(t\text{-LSK}(s)) \subset \mathcal{L}_{\mathrm{P}}(t\text{-SK}(s))$ *for* $t \geq 2$.

5 Conclusion

The study of the relation between PCGS and FRR was motivated by computational linguistics; both models seem to be useful in this field. While in [6] the basic relation between the computational power of these two models was established, the aim of this paper was to introduce and study the relevant complexity measures of PCGS and restrictions on computation of FRR.

We have succeeded in showing infinite hierarchies both for PCGSs and FRRs. The question of whether j-k-DDG is equal to $\mathcal{L}_{\mathrm{P}}(j\text{-LSK(k)})$ or not remains open.

We also believe that properly using nondeterminism the next conjecture can be shown.

Conjecture 1. For any $L \in$ j-k-DDG, there is a correctness preserving k-NSLnRR-automaton M with a left j-skeleton $SK(j)$ such that $L = L_P(M)$, and M has no auxiliary symbols outside of $SK(j)$.

References

1. J. Hromkovič, J. Kari, L. Kari, and D. Pardubská. Two lower bounds on distributive generation of languages. In *Proc. 19th International Symposium on Mathematical Foundations of Computer Science 1994*, *LNCS* vol. 841, Springer-Verlag, London, 423–432.
2. M. Lopatková, M. Plátek, and P. Sgall. Towards a formal model for functional generative description: Analysis by reduction and restarting automata. *The Prague Bulletin of Mathematical Linguistics* 87 (2007) 7–26.
3. V. Kuboň, M. Lopatková, M. Plátek, and P. Pognan. Segmentation of Complex Sentence. In:Lecture Notes In Computer Science 4188, 2006, 151-158.
4. F. Otto and M. Plátek. A two-dimensional taxonomy of proper languages of lexicalized FRR-automata. *Preproc. LATA 2008*, S.Z. Fazekas, C. Martin-Vide, and C. Tirnăucă (eds.), Tarragona 2008, 419 – 430.
5. D. Pardubská. Communication complexity hierarchy of parallel communicating grammar system. In: *Developments in Theoretical Computer Science*. Yverdon: Gordon and Breach Science Publishers, 1994. - 115–122. - ISBN 2-88124-961-2.
6. D. Pardubská and M. Plátek. Parallel Communicating Grammar Systems and Analysis by Reduction by Restarting Automata. Submitted to ForLing 2008.
7. Dana Pardubská, Martin Plátek, and Friedrich Otto. On the Correspondence between Parallel Communicating Grammar Systems and Restarting Automata. Technical Reports in Informatics, TR-2008-015, Comenius University, Bratislava(http://kedrigern.dcs.fmph.uniba.sk/reports/)
8. Gh. Păun and L. Santean. Parallel communicating grammar systems: the regular case. *Ann. Univ. Buc. Ser. Mat.-Inform.* 37 vol.2 (1989) 55–63.

Learning algorithms for small mobile robots: case study on maze exploration*

Stanislav Slušný and Roman Neruda and Petra Vidnerová

Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod vodárenskou věží 2, Prague 8, Czech Republic
slusny@cs.cas.cz

Abstract. *An emergence of intelligent behavior within a simple robotic agent is studied in this paper. Two control mechanisms for an agent are considered — new direction of reinforcement learning called relational reinforcement learning, and a radial basis function neural network trained by evolutionary algorithm. Relational reinforcement learning is a new interdisciplinary approach combining logical programming with traditional reinforcement learning. Radial basis function networks offer wider interpretation possibilities than commonly used multilayer perceptrons. Results are discussed on the maze exploration problem.*

1 Introduction

One of the key question of Artificial Intelligence is how to design intelligent agents. Several approaches have been studied so far. In our previous work, we have been examining mainly Evolutionary robotics (ER).

The ER approach attacks the problem through a self-organization process based on artificial evo-lu-ti-on [13]. Robot control system is typically realized by a neural network, which provides direct mapping from robot's sensors to effectors. Most of current applications use traditional multi-layer perceptron networks. In our approach we utilize local unit network architecture called radial basis function (RBF) network, which has competitive performance, more learning options, and (due to its local nature) better interpretation possibilities [18, 19].

This article gives summary of our experiences and comparison to Reinforcement Learning (RL) - another widely studied approach in Artificial Intelligence. RL is focusing on agent, that is interacting with the environment by its sensors and effectors. This interaction process helps agent to learn effective behavior. These kinds of tasks are commonly studied on miniature mobile robots of type Khepera [2] and E-puck [1].

2 Related work

The book [16] provides comprehensive introduction to the ER, with focus on robot systems. Recently, effort is made to study emergence of intelligent behavior within the group of robots.

Pioneering work was done by Martinoli [14]. He solved the task, in which group of simulated Khepera robots were asked to find "food items" randomly distributed on an arena. The control system was developed by the artificial evolution. Our work with single robot and robot teams were published in [19, 18].

Reinforcement learning is gaining increasing attention in recent years. The basic overview of the field can be found in [20].

3 Evolutionary robotics

The evolutionary algorithms (EA) [13, 12] represent a stochastic search technique used to find approximate solutions to optimization and search problems. They use techniques inspired by evolutionary biology such as mutation, selection, and crossover. The EA typically works with a population of *individuals* representing abstract representations of feasible solutions. Each individual is assigned a *fitness* that is a measure of how good solution it represents. The better the solution is, the higher the fitness value it gets. The population evolves toward better solutions. The evolution starts from a population of completely random individuals and iterates in generations. In each generation, the fitness of each individual is evaluated. Individuals are stochastically selected from the current population (based on their fitness), and modified by means of operators *mutation* and *crossover* to form a new population. The new population is then used in the next iteration of the algorithm.

Feed forward neural used as robot controllers are encoded in order to use them in the evolutionary algorithm. The encoded vector is represented as a floating-point encoded vector of real parameters determining the network weights.

Typical evolutionary operators for this case have been used, namely the uniform crossover and the mutation which performs a slight additive change in the parameter value. The rate of these operators is quite big, ensuring exploration capabilities of an evolutionary learning. A standard roulette-wheel selection is used together with a small elitist rate parameter. Detailed discussions about fitness function are presented in the next section.

* This work has been supported by the Ministry of Education of the Czech Republic under the project Center of Applied Cybernetics No. 1M684077004 (1M0567), S. Slušný been partially supported by by the Czech Science Foundation under the contract no. 201/05/H014G.

4 Relational reinforcement learning

The lack of theoretical insight into EA is the most serious problem of the previous approach. The RL is based on dynamic programming [6], which has been studied more than 50 years already. It has solid theoretical backgrounds built around Markov chains and several proved fundamental results. On the other side, it is not possible usually to fulfill theoretical assumptions in the experiments.

The general model of agent-environment interaction is modeled through the notion of rewards. The essential assumption of RL states, that agent is able to sense rewards coming from the environment. Rewards evaluate taken actions, agent's task is to maximize them. The next assumption is that agent is working in discrete time steps. Symbol S will denote finite discrete set of states and symbol A set of actions. In each time step t, agent determines its actual state and chooses one action. Therefore, agent's life can be written as a sequence

$$o_0 a_0 r_0 s_1 a_1 r_1 ... \tag{1}$$

where s_t denotes state, which is determined by processing sensors input, $a_t \in A$ action and finally symbol $r_t \in R$ represents *reward*, that was received at time t.

Formally, agent's task is to maximize

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... = \sum_{i=0} \gamma^i r_{t+i} \tag{2}$$

where the quantity $V^\pi(s_t)$ [16] is called discounted cumulative reward. It is telling us, what reward can be expected, if the agent starts in state s_t and follows policy π, $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards.

The most serious assumption of RL algorithms is the *Markov property*, which states, that agent does not need history of previous states to make decision. The decision of the agent is based on the last state s_t only. When this property holds, we can use theory coming from the field of *Markov decision processes* (MDP).

The policy π, which determines what action is chosen in particular state, can be defined as function $\pi : S \rightarrow A$, where $\pi(s_t) = a_t$. Now, the agent's task is to find optimal strategy π^*. Optimal strategy is the one, that maximalizes expected reward. In MDP, single optimal deterministic strategy always exists, no matter in what state has the agent started.

Optimal strategy π^* can now be defined as

$$\pi^* = \arg\max_\pi V^\pi(s), \forall s \in S \tag{3}$$

To simplify the notation, let's write $V^*(s)$ instead of symbol V^{π^*}, value function corresponding to optimal strategy π^*.

$$V^*(s) = \max_\pi V^\pi(s) \tag{4}$$

The first breakthrough of RL was the Q-learning algorithm [21, 4], which computes optimal strategy in described conditions.

The key idea of the algorithm is to define the so-called *Q-values*. $Q^\pi(s, a)$ is the expected reward, if the agent takes action a in state s and then follows policy π.

$$Q^\pi(s, a) = r(s, a) + \gamma V^\pi(s'), \tag{5}$$

where s' is the state, in which agent occurs taking action a in state s ($s' = \delta(s, a)$).

It is probably most commonly used algorithm of RL, mainly because of its simplicity. However, several improvements have been suggested to speed up the algorithm. In real life applications, state space is usually too big and convergence toward optimal strategy is slow. In recent years, there have been a lot of efforts devoted to rethinking idea of states by using function approximators [7], defining notion of options and hierarchical abstractions [5]. Relational reinforcement learning [11] is approach that combines RL with Inductive Logical Programming.

The distinction between classical RL and Relational Reinforcement Learning is the way how the Q-values are represented. In classical Q-learning algorithm are Q-values stored in the table. In relational version of the algorithm, they are stored in the structure called *Logical decision tree* [8]. In our experiments, we have used logical decision trees as implemented in the programs TILDE [8] from package ACE-ilProlog [9].

- for each s, a do
  - initialize the table entry $Q'(s, a) = 0$
  - $e = 0$
- do forever
  - $e = e + 1$
  - $i = 0$
  - generate a random state s_0
  - while not goal(s_i) do
    - select an action a_i and execute it
    - receive an immediate reward $r_i = r(s_i, a_i)$
    - observe the new state s_{i+1}
    - $i = i + 1$
  - endwhile
  - for $j = i - 1$ to 0 do
    - update $Q'(s_j, a_j) = r_j + \gamma \max_{a'} Q'(s_{j+1}, a')$

Fig. 1. Scheme of Q-learning algorithm, taken from [11].

5 Evolutionary RBF Networks

Evolutionary robotics combines two AI approaches: neural networks and evolutionary algorithms. Neural network receives input values from robot's sensors and it outputs control signals to the wheels. This way it realizes a control system of the robot.

Evolutionary algorithms [13, 12] are then used to train such a network. It would be difficult to utilize the training by traditional supervised learning algorithms since they require instant feedback in each step. Here we typically can evaluate each run of a robot as a good or bad one, but it is impossible to assess each one move as good or bad. Thus, the evolutionary algorithm represent one of the few possibilities, how to train the network.

The *RBF network* [17, 15, 10], used in this work, is a feed-forward neural network with one hidden layer of *RBF units* and linear output layer. The network function is given by Eq. (7).

$$y(\boldsymbol{x}) = \varphi\left(\frac{\parallel \boldsymbol{x} - \boldsymbol{c} \parallel}{b}\right) \tag{6}$$

$$f_s(\boldsymbol{x}) = \sum_{j=1}^{h} w_{js}\varphi\left(\frac{\parallel \boldsymbol{x} - \boldsymbol{c}_j \parallel}{b_j}\right), \tag{7}$$

where f_s is the output of the s-th output unit, y is the output of a hidden unit, φ is an activation function, typically Gaussian function $\varphi(s) = e^{-s^2}$.

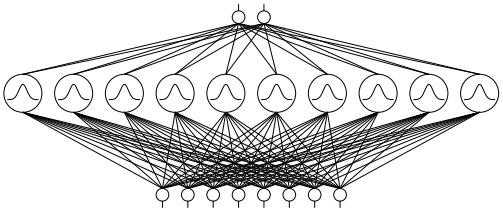

Fig. 2. A scheme of a Radial Basis Function Network.

The evolutionary algorithm is summarised in Fig. 3. It works with a population of *individuals* representing abstract representations of feasible solutions. Each individual is assigned a *fitness* that is a measure of how good solution it represents. The evolution starts from a population of completely random individuals and iterates in generations. Individuals are stochastically selected from the current population (based on their fitness), and modified by means of genetic operators *mutation* to form a new generation.

In case of RBF networks learning, each individual encodes one RBF network. The individual consists of h blocks:

$$I_{RBF} = \{B_1, \ldots, B_h\}, \tag{8}$$

1. START: Create population $P(0) = \{I_1, \cdots, I_N\}$.
2. FITNESS EVALUATION: For each individual evaluate fitness function.
3. TEST: If the stop criterion is satisfied, return the solution.
4. NEW GENERATION: Create empty population $P(i+1)$ and repeat the following procedure until $P(i+1)$ has N individuals.
   - **i)** Selection: Select two individuals from $P(i)$: $I_1 \leftarrow selection(P_i)$, $I_2 \leftarrow selection(P_i)$.
   - **ii)** Crossover: With probability p_c: $(I_1, I_2) \leftarrow crossover(I_1, I_2)$
   - **iii)** Mutation: With probability p_m: $I_k \leftarrow mutate(I_k), k = 1, 2$
   - **iv)** Insert: Insert I_1, I_2 into P_{i+1}
5. LOOP: Go to step 2.

Fig. 3. Scheme of an evolutionary algorithm.

where h is a number of hidden units. Each of the blocks contains parameter values of one RBF units:

$$B_k = \{c_{k1}, \ldots, c_{kn}, b_k, w_{k1}, \ldots, w_{km}\}, \tag{9}$$

where n is the number of inputs, m is the number of outputs, $\boldsymbol{c_k} = \{c_{k1}, \ldots, c_{kn}\}$ is the k-th unit's centre, b_k the width and $\boldsymbol{w}_k = \{w_{k1}, \ldots, w_{km}\}$ the weights connecting k-th hidden unit with the output layer. The parameter values are encoded using direct floating-point encoding.

We use standard *tournament selection*, *1-point crossover* and *additive mutation*. Additive mutation changes the values in the individual by adding small value randomly drawn from $\langle -\epsilon, \epsilon \rangle$.

The fitness function should reflect how good the robot is in given tasks and so it is always problem dependent. Detailed description of the fitness function is included in the experiment section.

6 Experiments

In order to compare performance and properties of described algorithms, we conducted simulated experiment. Miniature robot of type e-puck [1] was trained to explore the environment and avoid walls. E-puck is a mobile robot with a diameter of 70 mm and a weight of 50 g. The

Fig. 4. Miniature e-puck robot.

robot is supported by two lateral wheels that can rotate in both directions and two rigid pivots in the front and in the back. The sensory system employs eight "active infrared light" sensors distributed around the body, six on one side and two on other side. In "passive mode", they measure the amount of infrared light in the environment, which is roughly proportional to the amount of visible light. In "active mode" these sensors emit a ray of infrared light and measure the amount of reflected light. The closer they are to a surface, the higher is the amount of infrared light measured. The e-puck sensors can detect a white paper at a maximum distance of approximately 8 cm. Sensors return values from interval $[0, 4095]$. Effectors accept values from interval $[-1000, 1000]$. The higher value, the faster the motor is moving.

Sensor value	Meaning
0-50	NOWHERE
51-300	FEEL
301-500	VERYFAR
501-1000	FAR
1001-2000	NEAR
2001-3000	VERYNEAR
3001-4095	CRASHED

Table 1. Sensor values and their meaning.

Without any further preprocessing of sensor's and effector's values, the state space would be too big. Therefore, instead of raw sensor values, learning algorithms worked with "perceptions". Instead of 4095 raw sensor values, we used only 5 perceptions(table 1). Effector's values were processed in similar way: instead of 2000 values, learning algorithm chosen from values $[-500, -100, 200, 300, 500]$. To reduce the state space even more, we grouped pairs of sensors together and back sensors were not used at all.

The agent was trained in the simulated environment of size 100 x 60 cm and tested in more complex environment of size 110 x 100 cm. We used Webots [3] simulation software. Simulation process consisted of predefined number of steps. In each simulation step agent processed sensor values and set speed to the left and right motor. One simulation step took 32 ms.

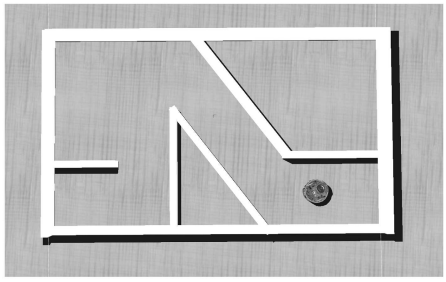

Fig. 5. Agent was trained in the simulated environment of size 100 x 60 cm.

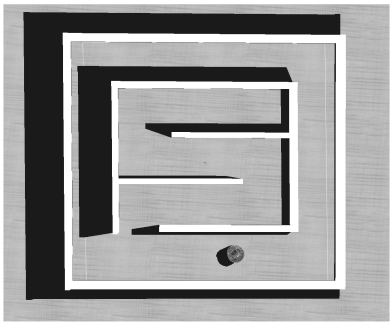

Fig. 6. Simulated testing environment of size 110 x 100 cm.

6.1 Evolutionary RBF networks

The evolutionary RBF networks were applied to the maze exploration task. The network input and output values are preprocessed in the same way as for the reinforcement learning.

To stimulate maze exploration, agent is rewarded, when it passes through the zone. The zone is randomly located area, which can not be sensed by an agent. Therefore, Δ_j is 1, if agent passed through the zone in j-th trial and 0 otherwise. The fitness value is then computed as

$$Fitness = \sum_{j=1}^{4}(S_j + \Delta_j), \tag{10}$$

where quantity S_j is computed by summing normalized trial gains $T_{k,j}$ in each simulation step k and trial j.

$$S_j = \sum_{k=1}^{800} \frac{T_{k,j}}{800}. \tag{11}$$

The three component $T_{k,j}$ motivates agent to move and avoid obstacles.

$$T_{k,j} = V_{k,j}(1 - \sqrt{\Delta V_{k,j}})(1 - i_{k,j}) \tag{12}$$

First component $V_{k,j}$ is computed by summing absolute values of motor speed in k-th simulation step and j-th trial, generating value between 0 and 1. The second component $(1 - \sqrt{\Delta V_{k,j}})$ encourages the two wheels to rotate in the same direction. The last component $(1 - i_{k,j})$ supports agent's ability to avoid obstacles. The value $i_{k,j}$ of the most active sensor in k-th simulation step and j-th trial provides a conservative measure of how close the robot is to an object. The closer it is to an object, the higher the measured value in range from 0 to 1. Thus, $T_{k,j}$ is in range from 0 to 1, too.

The experiment was repeated 10 times, each run lasted 200 generations (each generation corresponding to 800 simulation steps). In all cases the successful behavior was found, i.e. the evolved robot was able to explore the whole maze without crashing to the walls. See Fig. 7 for the mean, minimal and maximal fitness over 10 runs.

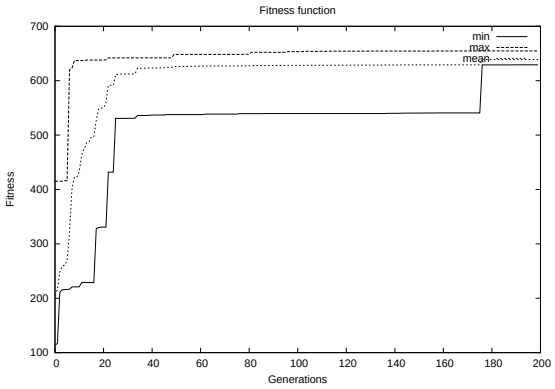

Fig. 7. The mean, minimal and maximal fitness function over 10 runs of evolution. Fitness is scaled in a way that successful walk through the whole maze corresponds to the fitness 600 and higher.

Table 2 and Figure 8 show parameters of an evolved network with five RBF units. For the sake of clarity, the parameters listed are also discretized. We can understand them as rules providing mapping from input sensor space to motor control. However, these 'rules' act in accord, since the whole network computes linear sum of the five corresponding gaussians.

Sensor			Width	Motor	
left	**front**	**right**		**left**	**right**
VERYNEAR	NEAR	VERYFAR	1.56	500	-100
FEEL	NOWHERE	NOWHERE	1.93	-500	500
NEAR	NEAR	NOWHERE	0.75	500	-500
FEEL	NOWHERE	NEAR	0.29	500	-500
VERYFAR	NOWHERE	NOWHERE	0.16	500	500

Table 2. Rules represented by RBF units (listed values are original RBF network parameters after discretization).

6.2 Reinforcement learning

The same experiment has been performed by means of relational reinforcement learning algorithm described above under the same simulated environment and identical conditions. The performance of the Reinforcement learning

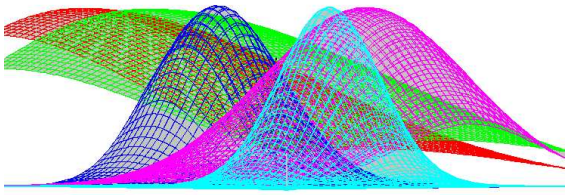

Fig. 8. The evolved RBF network (see also Tab. 2). Local units responses plotted in 2D input space corresponding to left and right sensory inputs.

agent is shown on figure 9. The graph shows average number of steps from each learning episode. It can be seen that after 10000 episodes, the agent has learned the successful behavior. This number roughly corresponds to the time complexity of the GA, where 200 populations of 50 individuals also result in 10000 simulations. The fitness of the solution found by RL is slightly better than the GA-found solution, on the other hand the inner representation of the neural network is much more compact.

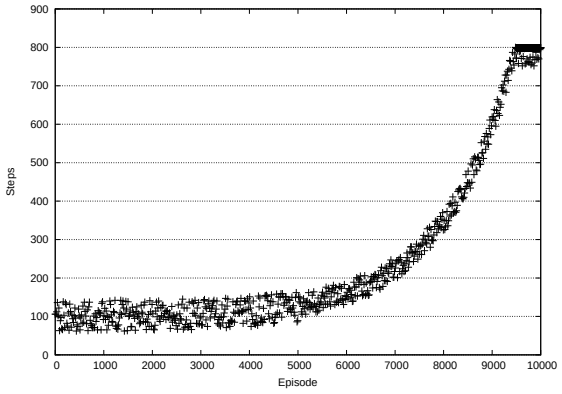

Fig. 9. Learning curve for Reinforcement Learning agent averaged on 10 runs.

7 Discussion

This article presented survey of popular approaches in mobile robotics used to robot behavior synthesis. In our future work, we would like to design hybrid intelligent system, combining the advantages of these approaches. This way, agent would benefit from using three widely studied fields: Inductive Logic Programming, Neural Networks and Reinforcement Learning. The Reinforcement Learning has strong mathematical background. On the other side, in real experiments, some of the assumptions are not realistic. Neural networks are very popular in robotics, because

they provide straightforward mapping from input signals to output signals, several levels of adaptation and are robust to noise. Inductive logic programming allows agent to reason about states, thus concentrating attention on the most promising parts of the state space.

The experiments showed that a preprocessing plays rather important role in the case of robotic agent control. In our approach we have chosen a rather strong processing of inputs and outputs, which is suitable for RL algorithms mainly. In our future work we would like to study control with less preprocessed inputs/outputs which can be used mainly for the neural network controller. Also, another immediate work is to extract the most frequently used state transitions from the RL algorithm and interpret them as rules in a similar fashion we did with the RBF network.

References

1. E-puck, online documentation. http://www.e-puck.org.
2. Khepera II documentation. http://k-team.com.
3. Webots simulator. http://www.cyberbotics.com/.
4. A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, pages 81–138.
5. A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. 13:341–379.
6. R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
7. D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming*. Ahtena Scientific, 1996.
8. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101:285–297, 1998.
9. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166.
10. D.S. Broomhead and D. Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.
11. S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning 43*, pages 7–52, 2001.
12. D. B. Fogel. *Evolutionary Computation: The Fossil Record*. MIT-IEEE Press, 1998.
13. J. Holland. *Adaptation In Natural and Artificial Systems*. MIT Press, reprinted edition, 1992.
14. A. Martinoli. *Swarm intelligence in autonomous Collective robotics: from tools to the analysis and synthesis of distributed control strategies.* Lausanne: Computer Science Department, EPFL, 1999.
15. J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:289–303, 1989.
16. S. Nolfi and D. Floreano. *Evolutionary Robotics — The Biology, Intelligence and Techology of Self-Organizing Machines.* The MIT Press, 2000.
17. T. Poggio and F. Girosi. A theory of networks for approximation and learning. Technical report, Cambridge, MA, USA, 1989. A. I. Memo No. 1140, C.B.I.P. Paper No. 31.
18. S. Slušný and R. Neruda. Evolving homing behaviour for team of robots. *Computational Intelligence, Robotics and Autonomous Systems. Palmerston North : Massey University*, 2007.
19. S. Slušný, R. Neruda, and P. Vidnerová. Evolution of simple behavior patterns for autonomous robotic agent. *System Science and Simulation in Engineering. - : WSEAS Press*, pages 411–417, 2007.
20. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
21. C. J. Watkings. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.