VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

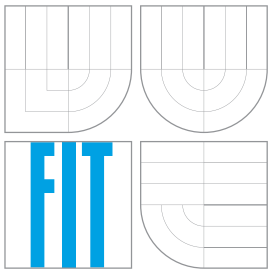
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN HRUŠKA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ ALGORITMY PRO PRÁCI S KONEČNÝMI AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN HRUŠKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ONDŘEJ LENGÁL
SUPERVISOR

BRNO 2013

Abstrakt

Výtah (abstrakt) práce v českém jazyce.

Abstract

Výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

Klíčová slova v českém jazyce.

Keywords

Klíčová slova v anglickém jazyce.

Citace

Martin Hruška: Efficient Algorithms for Finite Automata, bakalářská práce, Brno, FIT VUT v Brně, 2013

Efficient Algorithms for Finite Automata

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Lengála

.....
Martin Hruška
January 20, 2013

Poděkování

Rád bych tímto poděkoval vedoucímu této práce, Ing. Ondřeji Lengálovi, za odborné rady a vedení při tvorbě práce.

Contents

Chapter 1

Introduction

Finite automata is model of computation with applications in different fields, e.g., in compiler design, formal verification, theory of computation or language parsing. Finite automata recognize exactly set of regular languages [5]. In formal verification are often problems described as language inclusion [2]. Finite automata used for this purposes can have very large amount of states, so operations on them could be very computationally demanding. This fact leads to need of efficient implementation of algorithms for inclusion checking and the other operations on finite automata.

New algorithms for checking language inclusion were introduced [2], [4], but still not efficiently implemented for finite automata in laguage such as C++. These algorithms are based on pruning out the states, that aren't unnecessary for checking inclusion, so computation isn't so expensive. Algorithms with optimalization based on antichains and simulation from [2] are implemented in C++ library VATA [11], but only for tree automata. It's possible to use these algorithms for finite automata, which can be represented like one dimensional tree automata, but special implementation of algorithms for finite automata will be more efficient.

Main goal of these work is create extension of library VATA with efficient implementation of operation for finite algorithms, not only inclusion, but also union and intersection. After this introduction, in 2nd chapter of this document, will be defined theoretical background. Existing libraries for finite automata manipulation and library VATA will be introduced in chapter 3. Design of extentsion for VATA and used algorithms will take place in chapter 4. Implementation and optimalization is possible to find in chapter 5. Evaluation will be described in chapter 6 and final conclusion in chapter 7.

Chapter 2

Preliminaries

In this chapter will be defined theoretical bases for this thesis. No proofs are given, but they can be found in cited literature. Firstly, the finite automaton will be defined, then operation for regular languages and the other terms used in this work.

2.1 Finite automaton

Definition 2.1.1. Nondeterministic finite automata *(NFA) is a five-tuple* $N = (Q, \Sigma, \delta, s, F)$ *[6], where*

Q *is a finite set called states,*
Σ *is a finite set called alphabet,*
δ *is a function* $\delta : Q \times \Sigma \to Q$,. *We denote* $p \xrightarrow{a} q$ *transition from* $p \in Q$ *to* $q \in Q$ *under the input symbol* $a \in \Sigma$. *Function* δ *is called transition function.*
s *is state, that* $s \in Q$. s *is called start state.*
F *is set of states, that* $F \subset Q$. *Elements of* F *are called final states.*

2.2 Regular languages

2.2.1 Union, intersection, inclusion

Definition 2.2.1. *Let* L_1 *and* L_2 *be languages.* Union, intersection, inclusion *are defined [8] like:*

$Union: \ L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\}$
$Intersection: \ L_1 \cap L_2 = \{x | x \in L_1 \wedge x \in L_2\}$
$Inclusion: \ L_1 \subseteq L_2 \Leftrightarrow (\forall x)(x \in L_1 \Rightarrow x \in L_2)$

2.3 Antichain

Definition 2.3.1. *Let* S *be* partially order *set. Two elements* a *and* b, *where* $a \in S \wedge b \in S$, *are incomparable if neither* $a \leq b, nor \ b \leq a$. Antichain *is set* A, *where* $A \subset S: \ \forall (x, y) \in A$, x *and* y *are incomparable.*

2.4 Congruence relation

Definition 2.4.1. *Let G be group with n-ary operation O and sets of elements X.* Congruence *is equivalence relation R, which satisfies this condition:*

$$a_1 \sim_R b_1, a_2 \sim_R b_2, a_3 \sim_R b_3, \ldots, a_n \sim_R b_n \Rightarrow O_n(a_1, a_2, a_3, \ldots, a_n) \sim_R O_n(b_1, b_2, b_3, \ldots, b_n),$$

where $a_i \in X, b_i \in X$

2.5 Simulation

Antichain algorithm is based on pruning out unnecassary states for checking the language inclusion. For this state reduction algorithm needs to compute the simulation on the states of automata.

Definition 2.5.1. *A* Simulation *on $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is relation $\preceq \subseteq Q \times Q$ such that $p \preceq r$ only if (i) $p \in F \Rightarrow r \in F$ and (ii) for every transition $p \xrightarrow{a} p'$, there exists transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$ [10].*

Chapter 3

Existing finite automata libraries and VATA

3.1 Existing finite automata libraries

For finite automata manipulation exists many different libraries. Libraries have different purposes and are implemented in different languages, e.g. estabilished package of Java *dk.brics.automaton* [9], which is also reimplemented for C in *libfa* [7] and for C# in *Fare* [3]. Another libraries are *The RWTH FSA toolkit* [12] in C++, which also supports weighted automata, or *FSA Utilities toolbox* [1] in Prolog.

This is not definitely complete enumaration of finite automata libraries, but just few examples. Their main problem is that they do not contain the efficient inclusion testing. On the other side, new efficient algorithms introduced in [2] and in [4] have been for finite automata implemented only in OCaml and implementation in C++ could be much more efficient.

Because of these reasons, algorithms for finite automata manipulation will be implemented as extension of VATA (library for tree automata manipulation) in C++.

3.2 VATA

3.2.1 General

VATA is highly efficient open source library for manipulating non-deterministic tree automata. VATA is implemeted in C++ and uses Boosts C++ library. Informations and download of library can be found on its website [1] [11].

3.2.2 Design

VATA provides two kind of encoding for tree automata – Explicit Encoding (top-down) and Semi-symbolic encoding (top-down and bottom-up). There are supported these operations: union, intersection, elimination of unreachable states, inclusion checking, computation of simulation relation, language preserving size reduction based on simulation equivalence. Some operations are able only for specific encoding. Especially for inclusion checking (in

[1] http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

both forms – bottom-up and top-down) are used advanced optimalization because of improving performance. More details about implemented operations for tree automata are in [11].

Explicit Encoding

For storing explicit encoding top-down transitions is used *hierarchial data structure based on hash tables*. Inserting new transition to this structure requires a constant number of steps (exception is worst case scenario). For better performance is used *copy-on-write* technique [11].

Semi-symbolic Encoding

Transition functions in semi-symbolic encoding are stored in *multi-terminal binary desicion diagrams* (MTBDD), which are extension of *binary decision diagrams*. There are provided top-down and bottom-up representation of tree automata in semi-symbolic encoding [11].

3.2.3 Extension for finite automata

Main goal of this work is create extension of VATA for finite automata in explicit encoding. It is possible to use VATA for finite automata and represents finite automata like one dimensional, but special implementation will be more efficient. There will be used implementation of simulation computation from current VATA implementation.

Chapter 4

Design

4.1 Union

For union of two finite automaton is used algorithm from [8]. Pseudocode of union is in algorithm 1.

Algorithm 1: Automata union

Input: NFA's $\mathcal{A} = (Q_A, \Sigma_A, \delta_A, s_A, F_A)$, $\mathcal{B} = (Q_B, \Sigma_B, \delta_B, s_B, F_B)$
Output: NFA $\mathcal{C} = (Q_C, \Sigma_C, \delta_C, s_C, F_C)$, where $\{s_C, F_C\} \cap (Q_A \cup Q_B) \neq \varnothing$

1 $Q_C := Q_A \cup Q_B \cup \{s_C, F_C\}$;
2 $\Sigma_C := \Sigma_A \cup \Sigma_B$;
3 $\delta_C := \delta_A \cup \delta_B \cup \{s_C \xrightarrow{\epsilon} s_A, s_C \xrightarrow{\epsilon} s_B\} \cup \{f_i \xrightarrow{\epsilon} f : f_i \in F_A \vee f_i \in F_B\}$;

4.2 Intersection

For intersection is used algorithm based on [8]. Pseudocode could be found in algorithm 2.

Algorithm 2: Automata intersection

Input: NFA's $\mathcal{A} = (Q_A, \Sigma, \delta_A, s_A, F_A)$, $\mathcal{B} = (Q_B, \Sigma, \delta_B, s_B, F_B)$
Output: NFA $\mathcal{C} = (Q_C, \Sigma, \delta_C, s_C, F_C)$

1 $Q_C := Q_A \times Q_B$;
2 $s_C := (s_A, s_B)$;
3 $F_C := F_A \times F_B$;
4 $\delta_C := (p_1, q_1) \xrightarrow{a} (p_2, q_2)$, where $a \in \Sigma$, iff $p_1 \xrightarrow{a} q_1 \wedge p_2 \xrightarrow{a} q_2$;

4.3 Language inclusion

Language inclusion problem is desicion whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, where $\mathcal{L}(\mathcal{A})$ is language accepted by non-deterministic automaton $\mathcal{A}$ and so for $\mathcal{B}$. This problem is PSPACE-complete [2]. For decision of *language inclusion problem* exists some classical algorithms, but their problem is generating of states, which are not necessary for decision of inclusion problem.

This decreases efficient of the algorithm, so it is important to use some optimalization. In this work are used two approaches for optimalization, first uses antichains and simulations [2] for pruning states of NFA and second uses congruences [4].

For these algorithms let define post-image of production state:

Definition 4.3.1. $Post((p,P)) := \{(p',P') | \exists a \in \Sigma : (p,a,p') \in \delta, P' = \{p'' | \exists p \in P : (p,a,p'') \in \delta\}\}$

4.3.1 Checking inclusion with antichains

First approach uses optimalized method based on combination of antichains with simulation from [2]. Pseudocode is algorithm 3.

Algorithm 3: Language inclusion checking with antichains and simulations

Input: NFA's $\mathcal{A} = (Q_A, \Sigma, \delta_A, S_A, F_A)$, $\mathcal{B} = (Q_B, \Sigma, \delta_B, S_B, F_B)$.
A relation $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.

```
1  if there is an accepting product-state in {(s, S_B) | s ∈ S_A} then
2      return FALSE;
3  Proccesed:=∅;
4  Next:= Initialize({(s, Minimize(S_B)) | s ∈ S_A});
5  while (Next ≠ ∅) do
6      Pick and remove a product-state (r, R) from Next and move it to Processed;
7      forall the (p, P) ∈ {(r', Minimize(R')) | (r', R') ∈ Post((r, R))} do
8          if (p, P) is an accepting product-state then
9              return FALSE;
10         else
11             if ∄p' ∈ P s.t. p ⪯ p' then
12                 if ∄(x, X) ∈ Processed ∪ Next s.t. p ⪯ x ∧ X ⪯^{∀∃} P then
13                     Remove all (x, X) from Processed ∪ Next s.t. x ⪯ p ∧ P ⪯^{∀∃} X;
14                     Add (p, P) to Next;
15 return TRUE;
```

4.3.2 Checking inclusion with congruence

Checking inclusion with using congruence is based on *Hopcroft and Karp's* algorithm. In fact, algorithm serves for checking language equivalence, but it can be used also for checking laguage inclusion. Let X and Y be sets of states of NFA's. So $[X + Y] = [Y]$, iff $[X] \subseteq [Y]$. So it is possible to check equivalence for $X + Y$ and Y [4].

Optimalized algorithm uses *congruence closure function* c [4] for pruning out the unnecessary states for checking equivalence. Pseudocode is algorithm 4.

Algorithm 4: Language equivalence checking with congruence

Input: NFA's $\mathcal{A} = (Q_A, \Sigma, \delta_A, s_A, F_A),\ \mathcal{B} = (Q_B, \Sigma, \delta_B, s_B, F_B)$.

Output: TRUE, if $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{B})$ are in equivalence relation. Otherwise, FALSE.

```
1  Processed = ∅;
2  Next = ∅;
3  insert(s_A,s_B) into Next;
4  while Next ≠ ∅ do
5      extract(x,y) from Next;
6      if x,y ∈ c(Processed ∪ Next) then
7          skip;
8      if (x ∈ F_A ∧ y ∉ F_B) ∨ (x ∉ F_A ∧ y ∈ F_B) then
9          return FALSE;
10     insert({(x', y')|(x', y') ∈ post(x, y)}) in Next;
11     insert(x,y) in Processed;
12 return TRUE;
```

Chapter 5

Implementation

Chapter 6

Experimental evaluation

Chapter 7

Conclusion

Bibliography

[1] Fsa utilities toolbox. `http://odur.let.rug.nl/~vannoord/Fsa/`, 2005 [cit. 2013-01-19].

[2] A. Parosh Abdulla, Richard Mayr Lukáš Holík, Yu-Fang Chen, and Tomáš Vojnar. When simulation meets antichains (on checking language inclusion of nondeterministic finite (tree) automata). In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 6015, pages 158–174. Springer Verlag, 2010.

[3] Nikos Baxevanis. Fare. `https://github.com/moodmosaic/Fare`, 2012 [cit. 2013-01-19].

[4] Damien Pous Filippo Bonchi. Checking nfa equivalence with bisimulations up to congruence). Technical report, 2012.

[5] Rajeev Motwani John E. Hopcroft and Jeffrey D. Ullman. *Automata Theory, Languages, and Computation*. Pearson, 3 edition, 2007. ISBN 0-321-47617-4.

[6] Dexter Kozen. *Automata and Computability*. Springer, 1997. ISBN 0-387-94907-0.

[7] David Lutterkort. libfa. `http://augeas.net/libfa/index.html`, 2011 [cit. 2013-01-19].

[8] Alexander Meduna. *Automata and Languages*. Springer, 2000. ISBN 81-8128-333-3.

[9] Anders Moller. dk.brics.automaton. `http://www.brics.dk/automaton/`, 2011 [cit. 2013-01-19].

[10] Peter W. Kopke Monika R. Hezinger, Thomas A. Hezinger. Computing simulations on finite and infinite graphs. In *36th FOCS*, Annual Symposium on Foundations of Computer Science, pages 453–462. IEEE Computer Society Washington, 1995.

[11] Tomáš Vojnar Ondřej Lengál, Jiří Šimáček. Vata: A library for efficient manipulation of non-deterministic tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 79–94. Springer Verlag, 2012.

[12] Hermann Ney Stephan Kanthak. The rwth fsa toolkit. `http://www-i6.informatik.rwth-aachen.de/~kanthak/fsa.html`, 2005 [cit. 2013-01-19].