ACME++: Optimizing Stabilization Monoids Computations for Probabilistic Automata and the Star-Height Problem

Nathanaël Fijalkow[1,2], Hugo Gimbert[3], Edon Kelmendi[3], and Denis Kuperberg[4]

[1] LIAFA, Paris 7, France
[2] University of Warsaw, Poland
[3] LaBRI, Bordeaux, France
[4] Onera, Toulouse, France

Abstract. We present ACME++, a tool implementing part of the theory of regular cost functions over finite words. This theory is a quantitative extension of the classical theory of regular languages, which allows to express boundedness properties. We provide two applications: first to the classical theory of regular languages, by solving the star-height problem, and second to probabilistic automata, by implementing the Markov Monoid Algorithm. This tool is a follow-up to a prveious version, with optimized algorithms, we present a comparative study of their performances.

The dedicated webpage where the tool ACME++ can be downloaded is

http://www.liafa.univ-paris-diderot.fr/~nath/acme++.htm .

1 Introduction

Acme++ is a tool for deciding properties of some algebraic structures called stabilization monoids, with two motivations in mind:

- provide an effective tool to solve the starheight problem
- provide a tool to solve the value 1 problem for probabilistic automata.

The starheight problem

An excellent introduction about the starhieght probelm is given in [Kirsten], which mentions some of the important industrial applications as speech recognition [38], database theory [8], and image compression [4, 20]. This problem was considered as the most difficult problem in the theory of recognizable languages and it took 25 yers before being solved. Implementing Hashiguchi algorithm is hopeless: the algorithm proceeds by enumeration of certain expressions and has a terrible complexity [Lombardy + Sakarovitch]. It took another 22 years before an algorithm with a better algrithmic complexity was given by Kirsten.

Acme++ aims at solving the starheight problem for practical applications, albeit the doubly exponential space complexity of Kirsten algoithm is a challenge to tackle.

We use an optimization pointed out by Thomas Colcombet [**?**]: the structure of Subset Automata, whose algebraic properties allow minimization.

The value 1 problem

Probabilistic automata are a versatile tool widely used in speech recognition as well as a modelling tool for the control of systems with prtial observation (actually no observation). Probabilistic automata are natural extension on automata on finte words, with probabilstic transitions. The value 1 problem is natural when probabilistic automata are used as models of systems controlled by a blind controller, who is in charge of choosing the sequence of input letters in order to maximize the acceptance probability, see [LICS] for an introduction.

Stabilization semigroups are the key mathematical object needed to solve the two questions we are interested in. The starheight problem is related to the limitedness problem of distance atomata. A seminal paper by Simon "On semigroups of matrices over the tropical seimrings" gives a clean easy to read proof Then Kirsten showed how monoids could be used to solve the starheight problem using algebraic structures, giving an alternate proof to th every comple proof of Hashigushi, based on nested distance desert automata. Desert automata are automata with ranked counters that can be either incremented and reset, with the extra condition that if a counter is reset all counters of lower rank are reset as well. Colcombet extended further the work of Kirsten introducing stabilization monoids in order to handle general counter automata, without rank. The techniques of Simon were adapted to solve the value 1 problem for probabilistic automata.

Acme++ is a set of software tools to handle stabilization monoids, they can generate a stabilization monoid from its generators and test properties of these monoids, in particular the limitedness and the existance of value 1 witnesses.

Our results: provide a description of experimental results.

2 Stabilization Monoids for B- and Probabilistic Automata

The notion of stabilization monoids appears in two distinct contexts. It has first been developed in the theory of regular cost functions, introduced by Colcombet [Col09,Col13]. The underlying ideas have then been transferred to the setting of probabilistic automata [FGO12].

2.1 Stabilization Monoids in the Theory of Regular Cost Functions

At the heart of the theory of regular cost functions lies the equivalence between different formalisms: a logical formalism, cost MSO, two automata model, B- and S-automata, and an algebraic counterpart, stabilization monoids.

Here we briefly describe the model of B-automata, and their transformations to stabilization monoid. This automaton model generalizes the non-deterministic automata by adding a finite set of counters. Instead of accepting or rejecting a word as a non-deterministic automaton does, a B-automaton associates an integer value to each input word. Formally, a B-automaton is a tuple $\mathcal{A} = \langle A, Q, \varGamma, I, F, \varDelta \rangle$, where A is a finite alphabet, Q is a finite set of states, $\varGamma$ is a finite set of counters, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\varDelta \subseteq Q \times A \times \{\mathbf{ic}, \mathbf{e}, \mathbf{r}\}^{\varGamma} \times Q$ is the set of transitions.

A transition (p, a, τ, q) allows the automaton to go from state p to state q while reading letter a and performing action $\tau(\gamma)$ on counter γ. Action $\mathbf{ic}$ increments the current counter value by 1, $\mathbf{e}$ leaves the counter unchanged, and $\mathbf{r}$ resets the counter to 0.

The value of a run is the maximal value assumed by any of the counters during the run. The semantics of a B-automaton $\mathcal{A}$ is defined on a word w by $[\![\mathcal{A}]\!](w) = \inf\{\mathrm{val}(\rho) \mid \rho \text{ is a run of } \mathcal{A} \text{ on } w\}$. In other words, the automaton uses the non determinism to minimize the value among all runs. In particular, if $\mathcal{A}$ has no run on w, then $[\![\mathcal{A}]\!](w) = \infty$.

The main decision problem in the theory of regular cost functions is the limitedness problem. We say that a B-automaton $\mathcal{A}$ is *limited* if there exists N such that for all words w, if $[\![\mathcal{A}]\!](w) < \infty$, then $[\![\mathcal{A}]\!](w) < N$.

One way to solve the limitedness problem is by computing the stabilization monoid. It is a monoid of matrices over the semiring of counter actions $\{\mathbf{ic}, \mathbf{e}, \mathbf{r}, \omega\}^{\varGamma}$. There are two operations on matrices: a binary composition called product, giving the monoid structure, and a unary operation called stabilization. The stabilization monoid of a B-automaton is the set of matrices containing the matrices corresponding to each letter, and closed under the two operations, product and stabilization. As shown in [Col09,Col13], the stabilization monoid of a B-automaton $\mathcal{A}$ contains an unlimited witness if and only if it is not limited, implying a conceptually simple solution to the limitedness problem: compute the stabilization monoid and check for the existence of unlimited witnesses.

2.2 Stabilization Monoids for Probabilistic Automata

The notion of stabilization monoids also appeared for probabilistic automata, for the Markov Monoid Algorithm. This algorithm was introduced in [FGO12] to partially solves the value 1 problem: given a probabilistic automaton $\mathcal{A}$, does there exist $(u_n)_{n\in\mathbb{N}}$ a sequence of words such that $\lim_n \mathbb{P}_{\mathcal{A}}(u_n) = 1$?

Although the value 1 problem is undecidable, it has been shown that the Markov Monoid Algorithm correctly determines whether a probabilistic automaton has value 1 under the *leaktight* restriction. It has been recently shown that all classes of probabilistic automata for which the value 1 problem has been shown decidable are included in the class of leaktight automata [**?**], hence the Markov Monoid Algorithm is the *most correct* algorithm known to (partially) solve the value 1 problem.

As for the case of B-automata, the stabilization monoid of a probabilistic automaton is the set of matrices containing the matrices corresponding to each letter, and closed under the two operations, product and stabilization.

Note that the main point is that both the products and the stabilizations depend on which type of automata is considered, B-automata or probabilistic automata.

3 Computing the Stabilization Monoid of an Automaton

We report here on our implementation of the following algorithmic task:

We are given as input:

- A finite set of matrices S,
- A binary associative operation on matrices, the product, denoted $\cdot$,
- A unary operation on matrices, the stabilization, denoted $\sharp$.

The aim is to compute the closure of S under product and stabilization, called the stabilization monoid generated by S.

Note that if we ignore the stabilization operation, this reduces to computing the monoid generated by a finite set of matrices. It is well-known that this monoid can be exponential in the size of the matrices, so the crucial aspect here is space optimization. We describe two optimizations that allowed to significantly reduce space consumption.

Recall that in our application, the initial set of matrices is given by matrices M_a for $a \in A$, where A is the finite alphabet of the automaton. Hence we naturally associate to every element of the stabilization monoid a $\sharp$-expression: $(M_a \cdot M_b^\sharp \cdot M_a)^\sharp$ is associated to $(ab^\sharp a)^\sharp$. Many $\sharp$-expressions actually correspond to the same matrix: for instance, it may be that $(M_a \cdot M_a \cdot M_b)^\sharp = M_a^\sharp$.

The first optimization consists in storing a set R of rewriting rules between $\sharp$-expressions, in order to reduce as much as possible the number of products and stabilization.

A typical step in the algorithm is to construct a new $\sharp$-expression using newly constructed elements. Before computing the corresponding matrix, we check beforehand that this $\sharp$-expression cannot be reduced used the rewriting rules known so far:

- if it can be reduced, we avoid the costly computation of a product or a stabilization,
- it it cannot be reduced, then we compute the corresponding matrix, check whether it appeared already. If it did appear, then add a rewriting rule, otherwise add the matrix.

This means that the algorithm processes a lot of new $\sharp$-expressions, and should be able to quickly detect if a $\sharp$-expression can be rewritten using the set R. This task is performed very efficiently thanks to an optimized representation of R using pointers and a Hashing table.

The second optimization is about how matrices themselves are stored in memory. Empirically, although the stabilization monoid may contain a lot of different matrices, these matrices are very similar to each other. In particular, if we consider the set of lines of these matrices, it is rather small compared to the number of matrices. Hence the idea of storing lines in a table, and matrices as a set of pointers to the corresponding lines. This simple idea turned out to be a game changer.

The algorithm is presented in Algorithm 1.

```
Data: $S = \{(M_a, a) \mid a \in A\}$
Result: The stabilization monoid generated by $S$
Initialization: $\mathbf{New} = S$ and $R = \emptyset$ set of rewriting rules;
while $\mathbf{New} \neq \emptyset$ do
    // Closure by product;
    $\mathbf{New}_T = \mathbf{New}$;
    $\mathbf{New} = \emptyset$;
    for $(M, m) \in \mathbf{New}_T$ do
        for $(N, n) \in S$ do
            if $m \cdot n$ is not reducible using $R$ then
                Compute $M \cdot N$;
                if $(M \cdot N, w)$ is in $S$ for some $w$ then
                    Add $m \cdot n \to w$ to $R$;
                end
                else
                    Add $(M \cdot N, m \cdot n)$ to $\mathbf{New}_T$ and to $\mathbf{New}$;
                end
            end
        end
    end

    // Closure by stabilization;
    $\mathbf{New}_T = \mathbf{New}$;
    for $(M, m) \in \mathbf{New}_T$ do
        Compute $M^\sharp$;
        if $(M^\sharp, w)$ is in $S$ for some $w$ then
            Add $m^\sharp \to w$ to $R$;
        end
        else
            Add $(M^\sharp, m^\sharp)$ to $\mathbf{New}$;
        end
    end
    Add $\mathbf{New}$ to $S$;
end
return $S$;
```

Algorithm 1: Computing the stabilization monoid

References

[Col09] Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In *ICALP (2)*, pages 139–150, 2009.

[Col13] Thomas Colcombet. Regular cost-functions, part I: Logic and algebra over words. *Logical Methods in Computer Science*, 9(3), 2013.

[FGO12] Nathanaël Fijalkow, Hugo Gimbert, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. In *LICS*, pages 295–304, 2012.