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 manipulating stabilization monoids.
Acme++ can solve three fundamental algorithmic problems. First, compute the star height of a regular language, i.e. the minimal number of nested Kleene stars needed for expressing the language with a complement-free regular expression. Second, decide boundedness and equivalence for regular cost functions. Third, decide whether a probabilistic automaton has value 1, i.e. whether a probabilistic automaton accept words with probability arbitrarily close to 1.
All three problems reduce to the computation of the saturation monoid associated with an automaton, which is a challenge since the monoid is exponentially larger than the automaton. The compact data structures used in Acme++ allow this program to handle automata with several hundreds of states. This radically improves upon the performance of AcmeML, a previous version of the tool.

1 Introduction

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

- compute the star height of a regular language of finite words,
- decide boundedness and equivalence for regular cost functions,
- decide whether a probabilistic automaton has value 1.

The Star-height problem. The Star-height problem [Egg63] takes as input a regular language L and an integer h and decides whether there exists a regular expression for L with at most h nested Kleene stars. The minimal h having this property is called the star height of L. An excellent introduction about the Star-height problem is given in [Kir05], which mentions some of the important industrial applications as speech recognition [Moh97], database theory [GT01], and image compression [IK93,KMT04]. This problem was considered as one of the most difficult problems in the theory of recognizable languages and it took

25 years before being solved by Hashiguchi [Has88]. Implementing Hashiguchi algorithm is hopeless: even for a language L given via a 4-state automaton, a "very low minorant" of the number of languages to be tested with L for equality is $c^{(c^c)}$ with $c = 10^{10^{10}}$ [LS02]. It took another 22 years before an algorithm with a better algorithmic complexity was given by Kirsten in [Kir05]. Kirsten algorithm is based on the transformation of the input automaton on finite words to a nested counter automaton with the following key property: the counter automaton is *bounded* if and only if the star height of the regular language is less than h.

Acme++ aims at solving the Star-height problem for practical applications, albeit the doubly exponential space complexity of Kirsten's algorithm is a challenge to tackle. To our best knowledge, this is the first time a solution to the star-height problem is implemented.

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 partial observation or no observation. Probabilistic automata are a natural extension on automata on finite words, with probabilistic transitions, see [Rab63] for an introduction.

The Value 1 problem takes as input a probabilistic automaton on finite words and checks whether there are words accepted with probability arbitrarily close to 1, see [FGO12] for an introduction. The Value 1 problem is a reformulation of a natural controller synthesis problem: assume a blackbox finite state system with random events is controlled by a blind controller which inputs actions to the blackbox but has absolutely no feedback on the state of the system. Then the synthesis of controllers with arbitraily high reliability is equivalent to solving the value 1 problem.

Stabilization monoids. Stabilization monoids are the key mathematical object needed to solve both the Star-Height and the Value 1 problems: computing the starheight of a language can be done by computation of the stabilization monoid of nested distance desert automata and deciding whether a leaktight probabilistic automaton has value 1 reduces to the computation of the associated Markov monoid.

The Star-Height problem is related to the limitedness problem of nested distance automata. A seminal paper by Simon [Sim94] unifies several results using monoids of matrices, providing in particular a combinatorial tool called the forest factorization theorem. The first proof of the computability of the star height was given by Hashigushi and was very intricate. Thankfully, Kirsten gave a much more elementary and clean algorithm: a language has star-height at most h if and only if the stabilization monoid of an associated nested desert automaton contains an *unbounded witness*. Desert automata are automata with hierar-

chical counters that can be either incremented or 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 and generalizing regular language theory into cost function theory, including algebraic and logical characterizations [Col09,CKL10,Kup14]. Other problems than star-height can be reduced to boundedness question for regular cost functions (see e.g. [CL08a,CL08b,CKLV13]), and Acme++ includes a general-purpose cost functions library. The algebraic techniques of Simon were adapted to solve the value 1 problem for probabilistic automata: a leaktight automaton has value 1 if and only if its Markov monoid contains a value 1 witness [FGO12].

An efficient implementation. Computing stabilization monoids of automata is challenging since the construction induces an exponential blowup, and the two decision problems we tackle are PSPACE-complete. In order to find unbounded and value 1 witnesses in the stabilizaton monoids, Acme++ applies iteratively the product and stabilization operators of the monoid in order to generate all elements of the monoid, represented by a minimal collection of matrices on finite domains, and a set of rewrite rules that define how the product and the stabilization operators act on elements. Acme++ searches particular elements in the monoid and stops as soon as a witness is found. If no witness is found then the whole monoid is computed before the algorithm stops and outputs a negative answer.

To cope with large monoids, Acme++ uses compact data structures. In particular the matrices representing elements are not stored explicitely in memory: each vector appearing as a row or a column is stored at most once and matrices only store pointers to these rows and vertices. This way, Acme++ was able to generate monoids with more than fifty thousands elements and could handle the construction of stabilizaton monoid of automata with several hundreds of states.

The Starheight computation performed by Acme++ relies on an optimization from [CL08a]: the structure of Subset Automata, whose algebraic properties allow minimization. Moreover, the complexity of the algorithm is simply exponential because the input of Acme++ are restricted to deterministic automata, or non-deterministic automata for the complement language (dualized for uniformity of the input). We also use a heuristic inspired by experiments, where we start by testing potential witnesses that the input automaton has optimal loop complexity, in which case we can quickly compute an expression of optimal star-height.

The tool ACME++ is open source and available from Github, details are given on the webpage http://www.liafa.univ-paris-diderot.fr/~nath/?page=acmepp.

Related work. This tool is an improvement of a previous tool called Acme (or AcmeML) [FK14], implemented in OCaml, which was a first proof-of-

concept tool implementing regular cost functions algorithms and the Markov Monoid Algorithm. We introduced both optimizations of previous algorithms, and new functionalities such as the star-height computation. We provide quantitative experiments comparing both versions.

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 $\mathcal{A} = \langle A, Q, \Gamma, I, F, \Delta \rangle$, where A is a finite alphabet, Q is a finite set of states, Γ 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 $\Delta \subseteq Q \times A \times \{\mathbf{ic}, \mathbf{e}, \mathbf{r}\}^\Gamma \times Q$ 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 sets of counter ac-

tions $\{\mathbf{ic}, \mathbf{e}, \mathbf{r}, \omega\}^\Gamma$. There are two operations on matrices: a binary composition called product 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.

An unlimited witness will be in the form of a $\sharp$-expression, generated by the grammar $e := A|ee|e^\sharp$, for instance $a(ba^\sharp bba)^\sharp$. It describes the sequence of operations to perform in the monoid in order to reach the witness from the generators (the matrices associated with the alphabet), as well as how to build a sequence of words of unbounded value, by replacing $\sharp$ by n.

The notion of stabilization monoid can be abstracted, and any set equipped with these two laws (product and stabilization) and verifying certain axioms can be considered as a stabilization monoid, and can be used to recognize a cost function. In particular stabilization monoids can be minimized, which allows equivalence testing of regular cost functions.

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 solve 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 [FGKO15], 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. In particular the two frameworks do not seem to be reducible to each other at the level of automata, meaning there is no simple transformation from probabilistic automaton to B-automaton (or vice-versa) which preserves the stabilization monoid associated with the automaton.

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 allow 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 stabilizations.

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 using 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

4 Benchmarks

We compared the running times of Acme++ and the previous implementation AcmeML to justify this upgrade. We report the results in this section.

For the benchmarks we randomly chose automata which produce larger Markov monoids in order to observe the difference in performance between the two versions. The point of comparison is the size of the computed monoid rather than the number of states in the automaton, since some large automata can indeed produce small monoids, and vice-versa, some small automata can produce large monoids.

To this end, for each state s we pick a state t with uniform probability on the set of states and add a transition between s and t. After this we uniformly pick a number $p \in [0, 1]$, and for all other states t' different from t, we decide whether we will add a transition between s and t' by flipping a p-biased coin.

The results have been plotted in Fig. 4.

If for some sizes of the monoid in the x-axis there is no corresponding point for the AcmeML implementation, it means that it either timed out or had a stack overflow.

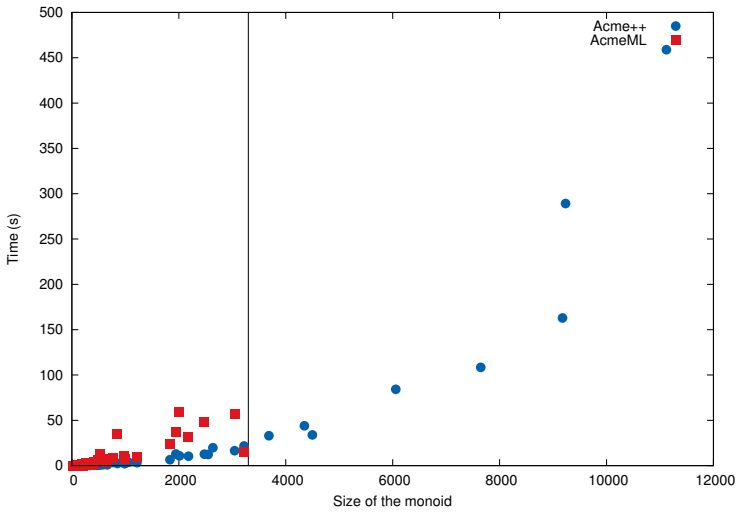

Fig. 1. The time it takes for the two implementations to calculate the Markov monoids generated by randomly picked automata of size 10

One can observe that there is a threshold in the size of the Markov monoid after which AcmeML will not be useful, i.e. it will either take too much time

or have a stack overflow. This threshold is expressed by the vertical line in the graph above (it hovers around 3500 elements).

Even on smaller examples Acme++ outperforms AcmeML by a considerable factor as seen in Fig. 4.

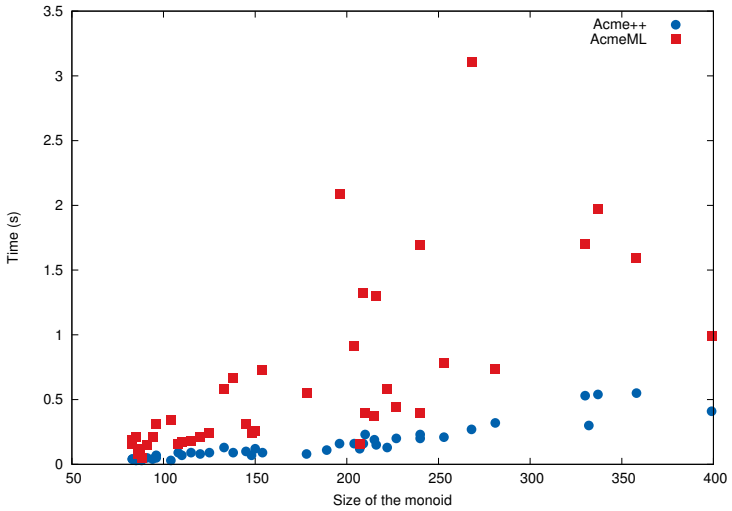

Fig. 2. The same experiment as in Fig.4 on automata generating smaller Markov monoids

5 The Star-height algorithm

The latest algorithm in the literature for computing star-height is designed for tree automata [CL08a], but we will use it here in the special case of words. The main improvement over the previous algorithm from [Kir05] is the identification of the structure of Subset Automata, which allows minimization.

We describe here briefly the ideas of the algorithm.

5.1 Subset automata

Definition 1. *[CL08a] A subset automaton $\mathcal{A}$ is a deterministic automaton with additional ϵ-transitions, such that ϵ-transitions encode a lattice structure on the state space of $\mathcal{A}$. More precisely, ϵ-transitions define a partial order on states, and for all states p, q, there are states x, y such that $x \xrightarrow{\epsilon} p, x \xrightarrow{\epsilon} q, p \xrightarrow{\epsilon} y$ and $q \xrightarrow{\epsilon} y$.*

Due to the algebraic nature of their definition, subset automata can be minimized in a similar way to deterministic automata.

Theorem 1. *[CL08a] Any language can be recognized by a subset automaton, obtained by a powerset construction from a non-deterministic automaton for the complement language.*

5.2 Reduction to limitedness

Let $\mathcal{A} = \langle A, Q, q_0, F_\mathcal{A}, \Delta_\mathcal{A} \rangle$ be a subset automaton for a language L, and $k \in \mathbb{N}$. We recall that there are ϵ-transitions, so $\Delta_\mathcal{A} \subseteq Q \times (A \cup \{\epsilon\}) \times Q$.

We build a B-automaton $\mathcal{B}$ with $k+1$ counters $\gamma_0, \gamma_1, \ldots, \gamma_k$, and states $Q_\mathcal{B} = \bigcup_{i=1}^{k+1} Q^i$ that we view as a subset of Q^*.

Let $j \in [0, k]$, we will note R_j the counter action performing $\mathbf{r}$ on counters γ_p with $p \geq j$ and $\mathbf{e}$ on counters γ_p with $p < j$. Simlarly, we will note I_j the counter action performing $\mathbf{r}$ on counters γ_p with $p > j$, $\mathbf{ic}$ on counter γ_j, and $\mathbf{e}$ on counters γ_p with $p < j$. We finally note $\mathbf{e}$ the action performing $\mathbf{e}$ on all counters. These particular counter actions are called *hierarchical*, they correspond to the nested distance automata from [Kir05]. Their advantage over general actions on multiple counters is that there is a total order or preference, namely $R_0 \leq R_1 \leq \cdots \leq R_k \leq \mathbf{e} \leq I_0 \leq I_1 \leq \cdots \leq I_k$. This greatly simplifies the shape of elements of the stabilization monoid, since for each pair of states (q_i, q_j) it suffices to store the best possible action to go from q_i to q_j instead of the set of all available actions.

The automaton $\mathcal{B}$ is defined as follows:

- The initial state is q_0 (word of length 1).
- A state wp is final if and only if $p \in F_\mathcal{A}$.
- If $(p, a, q) \in \Delta_\mathcal{A}$ and $w \in Q^{\leq k}$, there is a transition $wp \xrightarrow{a:I_{|w|}} wq$ in $\mathcal{B}$. If $a = \epsilon$, the action may equivalently be replaced by $\mathbf{e}$, as we do in the implementation.
- If $w \in Q^{\leq k-1}$ and $p \in Q_\mathcal{A}$, there is a transition $wp \xrightarrow{\epsilon:R_{|wp|}} wpp$ in $\mathcal{B}$.
- If $w \in Q^{\leq k-1}$ and $q, p \in Q_\mathcal{A}$, there is a transition $wpq \xrightarrow{\epsilon:R_{|wp|}} wq$ in $\mathcal{B}$.

The following theorem guarantees the correctness of the reduction.

Theorem 2. *[CL08a] The automaton $\mathcal{B}$ is limited if and only if L is expressible with a regular expression of star-height k.*

Therefore, for any fixed k, we can decide in EXPSPACE whether a regular language has star-height k, if it is given via a nondeterministic automaton for the complement (or via a deterministic automaton).

We are given as input an integer k and the dual of the non-deterministic automaton for $A^* \setminus L$. In particular a deterministic automaton for L works.

1. Build a subset automaton $\mathcal{A}$ via a powerset construction (exponential space),
2. compute the minimal subset automaton $\mathcal{A}_m$ of L from $\mathcal{A}$,
3. build the automaton $\mathcal{B}$ as above,
4. check $\mathcal{B}$ for limitedness using the stabilization monoid algorithm.

Let n be the size of the input automaton. The automaton $\mathcal{B}$ is of size $O(2^{nk})$, and moreover the stabilization monoid is exponential in the size of $\mathcal{B}$. Therefore, our algorithm runs in space $O(2^{2^{nk}})$. However, the limitedness step could be performed in PSPACE using on-the-fly computation on the monoid, so the theoretical complexity of the problem is EXPSPACE for an input of this shape. Note that Kirsten's algorithm uses doubly exponential space because it takes a non-deterministic automaton for L as input.

5.3 Loop complexity heuristic

Given an automaton $\mathcal{A}$, its *loop complexity* $\mathrm{LC}(\mathcal{A})$ is the optimal star-height of an expression obtain from it via standard algorithms [Egg63]. Moreover, the star-height of a language L is equal to $\min\{\mathrm{LC}(\mathcal{A}) | L(\mathcal{A}) = L\}$.

There are many natural cases where the star-height is equal to the loop-complexity of the input automaton [Coh70], for instance if all transition labels are distincts. These automata are relevant for instance for modelling processes where communication between two components (states) are uniquely identified.

Since computing the loop-complexity is exponentialy faster than the star-height algorithm, we start by computing "for free" an expression witnessing the loop-complexity.

Additionnally, we argue we can use this expression as a speed-up for the main algorithm, in two different ways. First, the loop-complexity provides an upper bound for star-height, and we can therefore stop the search if all heights strictly lower have been tested. The second way is heuristic: during the tests of Acme++, it seemed that such expressions witnessing loop complexity could often be used as unboundedness witnesses in the computed stabilization monoid, in cases where the loop complexity does not match the star-height.

Moreover, having computed this expression allows us to provide a witness of optimal star-height when it matches the loop-complexity. In general, the star-height algorithm does not provide any witness.

This naturally led us to implement a heuristic using these locally optimal expressions. When testing star-height h for an input automaton $\mathcal{A}$ (starting with $h = 0$), we execute the following steps:

- Compute a regular expression e matching $\mathrm{LC}(\mathcal{A})$.
- If $\mathrm{LC}(\mathcal{A}) = h$, then return h together with e and stop.
- Otherwise, turn e into a $\sharp$-expression e' by removing disjunctions.
- Test e' as unboundedness witness in the stabilization monoid.
  - If e' is such a witness, increase h, and go back to Step 1,
  - Otherwise, test unboundedness, and increase h if a witness is found.

References

[CKL10] Thomas Colcombet, Denis Kuperberg, and Sylvain Lombardy. Regular temporal cost functions. In *ICALP (2)*, pages 563–574, 2010.

[CKLV13] Thomas Colcombet, Denis Kuperberg, Christof Löding, and Michael Vanden Boom. Deciding the weak definability of büchi definable tree languages. In *CSL*, pages 215–230, 2013.

[CL08a] Thomas Colcombet and Christof Löding. The nesting-depth of disjunctive μ-calculus for tree languages and the limitedness problem. In *CSL, Bertinoro, Italy, September 16-19, 2008*, pages 416–430, 2008.

[CL08b] Thomas Colcombet and Christof Löding. The non-deterministic mostowski hierarchy and distance-parity automata. In *ICALP (2)*, pages 398–409, 2008.

[Coh70] Rina S. Cohen. Star height of certain families of regular events. *J. Comput. Syst. Sci.*, 4(3):281–297, 1970.

[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.

[Egg63] L. C. Eggan. Transition graphs and the star-height of regular events. *Michigan Math. J.*, 10:385–397, 1963.

[FGKO15] Nathanaël Fijalkow, Hugo Gimbert, Edon Kelmendi, and Youssouf Oualhadj. Deciding the value 1 problem for probabilistic leaktight automata. *Logical Methods in Computer Science*, 11(1), 2015.

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

[FK14] Nathanaël Fijalkow and Denis Kuperberg. ACME: automata with counters, monoids and equivalence. In *ATVA 2014, Sydney, NSW, Australia*, pages 163–167, 2014.

[GT01] Gösta Grahne and Alex Thomo. Approximate reasoning in semistructured data. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), Rome, Italy, September 15, 2001*, 2001.

[Has88] Kosaburo Hashiguchi. Algorithms for determining relative star height and star height. *Inf. Comput.*, 78(2):124–169, 1988.

[IK93] Karel Culik II and Jarkko Kari. Image compression using weighted finite automata. *Computers & Graphics*, 17(3):305–313, 1993.

[Kir05] Daniel Kirsten. Distance desert automata and the star height problem. *ITA*, 39(3):455–509, 2005.

[KMT04] Frank Katritzke, Wolfgang Merzenich, and Michael Thomas. Enhancements of partitioning techniques for image compression using weighted finite automata. *Theor. Comput. Sci.*, 313(1):133–144, 2004.

[Kup14] Denis Kuperberg. Linear temporal logic for regular cost functions. *Logical Methods in Computer Science*, 10(1), 2014.

[LS02] Sylvain Lombardy and Jacques Sakarovitch. Star height of reversible languages and universal automata. In *LATIN: Theoretical Informatics, Cancun, Mexico, April 3-6, 2002*, pages 76–90, 2002.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[Rab63] Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[Sim94] Imre Simon. On semigroups of matrices over the tropical semiring. *ITA*, 28(3-4):277–294, 1994.