DRAFT: Go Implementation of Up-To Techniques for Equivalence of Weighted Languages

Alessandro Cheli
Undergraduate Student
Department of Computer Science
Università di Pisa
Pisa, PI 56127
a.cheli6@studenti.unipi.it

October 5, 2020

Abstract

Weighted automata generalize non-deterministic automata by adding a quantity expressing the weight (or probability) of the execution of each transition. In this work we propose an implementation of two algorithms for computing language equivalence in finite state weighted automata (WAs). The first, a linear partition refinement algorithm, computes the largest linear weighted bisimulation for any given LWA (Linear Weighted Automaton) through an iterative method, the second algorithm checks the language equivalence of two vectors (states) for a given weighted automata by using an additional data structure representing a congruence relation. We then compare results of the two algorithms to verify their correctness and performance on randomly generated samples. We finally provide comparison of runtime statistics and suggest which of the two algorithm is the best choice for different usage cases.

***K*eywords** First keyword · Second keyword · More

1 Introduction

TODO da fare

Checking whether two nondeterministic finite automata (NFA) accept the same language is important in many application domains such as compiler construction and model checking. Unfortunately, solving this problem is costly: it is PSPACE-complete (TODO cita)

In [1], up-to techniques are defined for weighted systems over arbitrary semirings, while in [2], up-to techniques are defined for Linear Weighted Automata (LWAs), under a more abstract coalgebraic perspective.

A well detailed example of the comparison of algorithms to compute language equivalence, precisely between `HKC` and an alternative algorithm called the *antichain algorithm* ([3]), was published in 2017 [4].

2 Preliminaries and Notation

Note. Given two vector spaces V_1, V_2 we write $V_1 + V_2$ to denote $\text{span}\left(V_1 \cup V_2\right)$

Definition 2.1. A *weighted automaton* over a field $\mathbb{K}$ and an alphabet A is a triple (X, o, t) such that X is a finite set of states, $t = \left\{t_a : X \to \mathbb{K}^X\right\}_{a \in A}$ is a set of transition functions indexed over the symbols of the alphabet A and $o : X \to \mathbb{K}$ is the output function. The transition functions will be represented as $X \times X$ matrices. A^* is the set of all words over A, more precisely the free monoid with string concatenation as the monoid operation and the empty word ϵ as the identity element. We denote with aw the concatenation of a symbol a to the word $w \in A^*$. A weighted language is a function $\psi : A^* \to \mathbb{K}$. A function mapping each state vector into its accepted language, $[\![\cdot]\!] : \mathbb{K}^X \to \mathbb{K}^{A^*}$ is defined as follows for every weighted automaton:

$$\forall v \in \mathbb{K}^X, a \in A, w \in A^* \qquad [\![v]\!](\epsilon) = o(v) \qquad [\![v]\!](aw) = [\![t_a(v)]\!](w)$$

Two vectors $v_1, v_2 \in \mathbb{K}^{X \times 1}$ are called weighted language equivalent, denoted with $v_1 \sim_l v_2$ if and only if $[\![v_1]\!] = [\![v_2]\!]$. One can extend the notion of language equivalence to states rather than for vectors by assigning to each state $x \in X$ the corresponding unit vector $e_x \in \mathbb{K}^X$. When given an initial state i for a weighted automaton, the language of the automaton can be defined as $[\![i]\!]$.

Definition 2.2. A binary relation $R \subseteq X \times Y$ between two sets X, Y is a subset of the cartesian product of the sets. A relation is called *homogeneous* or an *endorelation* if it is a binary relation over X and itself: $R \subseteq X \times X$. In such case, it is simply called a binary relation over X. An *equivalence relation* is a binary relation that is reflexive, symmetric and transitive.

Definition 2.3. The congruence closure $c(R)$ of a relation R is the smallest congruence relation R' such that $R \subseteq R'$

An equivalence relation which is compatible with all the operations of the algebraic structure on which it is defined on, is called a *congruence relation*. Compatibility with the algebraic structure operations means that algebraic operations applied on equivalent elements will still yield equivalent elements.

We omit the coalgebraic definition for *linear weighted automata* seen in [2] and give a more intuitive definition, which fits our implementation when $\mathbb{K} = \mathbb{R}$. In this implementation, we focus only on weighted automata defined over the field of real numbers $\mathbb{R}$.

Definition 2.4. A *linear weighted automaton* (in short, LWA) over the field $\mathbb{K}$ and an alphabet A is a triple $L = (V, o, \{t_a\}_{a \in A})$ where V is a vector space representing the state space, $o : V \to \mathbb{K}$ is a linear map associating to each state its output weight, and $t = \{t_a = V \times V\}_{a \in A}$ is the set of transition functions, represented with liner maps that for each input $a \in A$ associate the next state, in this case a vector in V. As in [5], we have that $\dim(L) = \dim(V)$.

Given a weighted automaton, one can build a corresponding linear weighted automaton by considering the free vector space generated by the set of states X in the WA, and by linearizing o and t. If X is finite, as in our implementations of the algorithms, we can use the same matrices for t and o in both the WA and the corresponding LWA. We are only considering a finite number of states and therefore finite dimensional vector spaces. Let n be the number of states in an WA. We have that in the corresponding LWA, the transition functions t_a are still represented as $\mathbb{K}^{n \times n}$ matrices. $o \in \mathbb{K}^{1 \times n}$ is represented as a row vector. $t_a(v)$ denotes the vector obtained by multiplying the matrix t_a by the column vector $v \in \mathbb{K}^{n \times 1}$. $o(v)$ denotes the scalar $s \in \mathbb{K}$ obtained by dot product of the row vector o with $v \in \mathbb{K}^{n \times 1}$.

Definition 2.5. The language recognized by a vector $v \in V$ of an LWA (V, o, t) is defined for all words $w \in A*$ as $[\![v]\!]_V^{\mathcal{L}}(w) = o(v_n)$ where v_n is the vector reached from v through the composition of the transition functions corresponding to the words in w.

$$[\![v]\!]_V^{\mathcal{L}}(w) = \begin{cases} o(v) & \text{if } w = \epsilon \\ [\![t_a(v)]\!]_V^{\mathcal{L}}(w') & \text{if } w = aw' \end{cases}$$

We define $\approx_{\mathcal{L}}$ as the behavioral equivalence for a given LWA (V, o, t) as

$$\forall v_1, v_2 \in V,\ v_1 \approx_{\mathcal{L}} v_2 \iff [\![v_1]\!]_V^{\mathcal{L}} = [\![v_2]\!]_V^{\mathcal{L}} \tag{1}$$

Proof is available in section 3.3 of [2]

Language equivalence can be now expressed in terms of linear weighted bisimulations (LWBs for short). Differently from weighted bisimulations, LWBs can be seen both as relations and as subspaces. The subspace representation of LWBs is used in the backwards partition refinement algorithm implemented in [2] and in this work.

Definition 2.6. *Linear Relations:*
Let U be a subspace of V. The binary relation R_U over V is defined by

$$v_1\ R_U\ v_2 \qquad \iff \qquad v_1 - v_2 \in U$$

The relation R is linear if there exists a subspace U such that $R \equiv R_U$. A linear relation is a total equivalence relation on V.

Definition 2.7. *Kernel of a Relation and Linear Extension*
Let R be a binary relation over V. The *kernel* of R, is the set $\ker(R) = \{v_1 - v_2 \mid v_1\ R\ v_2\}$. The *linear extension* of R, written as R^ℓ, is defined by

$$v_1\ R^\ell\ v_2 \quad \iff \quad (v_1 - v_2) \in \operatorname{span}(\ker(R))$$

Lemma 2.1. *Let U be a subspace of V, then $\ker(R_U) = U$*

Definition 2.8. *Linear Weighted Bisimulation:*
Let (V, o, t) be a linear weighted automaton. A linear relation $R \subseteq V \times V$ is a *linear weighted bisimulation* if $\forall (v_1, v_2) \in R$ it holds that:

1. $o(v_1) = o(v_2)$
2. $\forall a \in A,\ t_a(v_1)\ R\ t_a(v_2)$

Lemma 2.2. *Let (V, o, t) be a linear weighted automaton. A linear relation R over V is a linear weighted bisimulation if and only if*

1. $R \subseteq \ker(o)$
2. R is t_a-invariant $\forall a \in A$

Theorem 3 in section 3.3 of [2], states that $\ker\left([\![-]\!]_V^{\mathcal{L}}\right)$ is the largest linear weighted bisimulation on V. As a corollary, we obtain that $\approx_{\mathcal{L}}$ is the largest linear weighted bisimulation.

We now introduce a lemma that will be fundamental in the next sections of this work.

Lemma 2.3. $\approx_{\mathcal{L}}$ *coincides with* $\sim_l$*:*

Let (X, o, t) be a WA and $(\mathbb{K}^X, o^\sharp, t^\sharp)$ the corresponding linear weighted automaton. Then $\forall x \in X,\ \ [\![x]\!] = [\![x]\!]_{\mathbb{K}^X}^{\mathcal{L}}$

3 Algorithms

The first algorithm we implement to compute language equivalence, called `HKC`, is adapted from [1]. It was first introduced by Bonchi and Pous in [6]. The algorithm, extending the Hopcroft and Karp procedure [7] with *congruence closure*, is proven to be sound and complete [1]. Originally, the algorithm was defined for WAs over semirings, but we recall that in this work we are only considering fields, in particular the field of real numbers ($\mathbb{K} = \mathbb{R}$). The problem of checking language equivalence has been proven undecidable for an arbitrary semiring, so termination may not always be guaranteed. However, it has been shown to be decidable for a broad range of semirings, for example, all the complete and distributive lattices. `HKC` computes $v_1 \sim_l v_2$ for a given weighted automaton $W = (X, t, o)$ and two vectors $v_1, v_2 \in \mathbb{K}^X$. by computing a congruence closure, and it does so without linearizing the state space.

We compare `HKC` with an algorithm called *Backwards Partition Refinement*, that we will call BPR for short.

(TODO chiedi a filippo: nelle conclusioni di [2] dice che è diverso dall'algoritmo visto nel paper di boreale [5]. è il caso anche per i field e i numeri reali???? o sono lo equivalenti?).

Adapted from [2], `BPR` is a fixed-point iterative method that, given an LWA $L = (V, t, o)$, computes a basis of the subspace of V representing the complementary relation of $\approx_{\mathcal{L}}$ (we later show it to be the orthogonal complement in case V is an inner product space). Another version of the algorithm is defined in the same work, called *Forward Partition Refinement*, which directly computes a basis for $\approx_{\mathcal{L}}$ but is shown to be way less efficient than the backwards version.
Note. Recall from section 2 that $\approx_{\mathcal{L}}$ is a linear relation, therefore $v_1 \approx_{\mathcal{L}} v_2 \iff (v_1 - v_2) \in \ker(\approx_{\mathcal{L}})$

The `BPR` algorithm starts from a relation R_0, that is the complement of the relation identifying vectors with equal weights. It then incrementally computes the space of all states that are reachable from R_0 in a *backwards* direction. Intuitively, "going backwards" means working with the transpose transitions functions t_a^t.

In the next section we compare execution results of our implementation of the algorithms `BPR` and `HKC` to verify the correctness of the implementation, and to analyze when one of the two algorithms is the convenient choice. Lemma 2.3, introduced above, is key to our work. By stating that $\approx_{\mathcal{L}}$ coincides with $\sim_l$, we can confidently say that the two algorithms compute an answer for same the decision problem:

Are two vectors v_1 and v_2 language-equivalent for a given weighted automata?

TODO costo computazionale HKC. BPR has a cost of $O(n^4)$ operations to initially compute the largest linear weighted bisimulation, which can be eventually reduced to $O(n^3)$. In our implementation, by computing a basis of the orthogonal complement of $\approx_{\mathcal{L}}$, the cost of checking if two vectors are language equivalent is reduced to the cost of matrix multiplication ($O(n^2)$). BPR is efficient when we have to decide if a large number of vectors in a WA are language equivalent.

3.1 HKC Algorithm

We give a pseudocode definition of the HKC procedure:

Figure 1: The HKC(v_1, v_2) procedure

```
HKC(v1, v2):
R := ∅; todo := ∅
while todo is not empty do
    extract (v1', v2') from todo
    if (v1', v2') ∈ c(R) then continue
    if o(v1') ≠ o(v2') then return false
    for all  a ∈ A
        insert (t_a(v1'), t_a(v2')) into todo
    insert (v1', v2') into R
return true
```

3.2 Backwards Partition Refinement Algorithm for the Largest Weighted Bisimulation

We now recall the backwards algorithm for computing $\approx_{\mathcal{L}}$ defined in [2]. The algorithm is defined by the iterative method:

$$R_0 = \ker(o)^0, \qquad R_{i+1} = R_i + \sum_{a \in A} t_a^t(R_i) \tag{2}$$

Where $\ker(o)^0$ is an annihilator. The algorithm stops when $R_{j+1} = R_j$. An index $j \leq \dim(V)$ is guaranteed to exist, such that the algorithms terminates at step j. It follows that $\approx_{\mathcal{L}} = R_j^0$. Proof is available in section 4.2 of [2]

4 Implementation

The algorithms and data structures for this paper are implemented in the Go programming language. This implementation makes use of the Gonum library for numerical computations. We only import the Gonum libraries for matrices and linear algebra and visual plotting of samples and functions. Real numbers are implemented with double precision floating point numbers, known as the `float64` type in the Go programming language.

Definition 4.1. Applications of SVD

Let's consider the singular value decomposition of a matrix $A \in \mathbb{R}^{m \times n}$:

$$A = U\Sigma V^T \qquad \Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r) \qquad U \in \mathbb{R}^{m \times m} \qquad V \in \mathbb{R}^{n \times n}$$

Where V and U are orthogonal and the singular values are ordered: $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq 0$. It follows that rank (A) is equal to the number of nonzero singular values, and as explained in [8]:

1. rank $(A) =$ rank $(\Sigma) = r$
2. The column space of A is spanned by the first r columns of U.
3. The null space of A is spanned by the last nr columns of V.
4. The row space of A is spanned by the first r columns of V.
5. The null space of A^T is spanned by the last mr columns of U.

Of our interest, are only the computation of the null space and columns space. The implementation, applying SVD, can be found in files `lin/colspace.go` and `lin/nullspace.go`.

4.1 Implementing the backwards partition refinement algorithm

To compute $\approx_{\mathcal{L}}$ at the last step of the algorithm, we need to compute R_j^0. If V is a vector space and W is a subspace of W, the annihilator of W, respectively W^0 is a subspace of the space V^* of linear functionals on V. W^0 are the functionals that annihilate on W. Since we are working on subspaces of $\mathbb{R}^n$, we can directly compute the orthogonal complement in our implementation instead of the annihilator.

Proposition 4.1. *If V is a finite dimensional vector space defined with an inner product $\langle \cdot, \cdot \rangle$ and W is a subspace of V then the image of the annhilitaor W^0 through the linear isomorphism $\varphi : V^* \to V$ induced by the inner product, is the orthogonal of W with respect to the said inner product.*

Proof. Let V be an inner product space over the field $\mathbb{K}$ with an inner product defined as $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{K}$. Every linear functional can be represented with a vector. Let $\xi : V \to \mathbb{K}$ be a functional, $\xi \in W^0$. Because $\xi(w) = 0 \quad \forall w \in W$, if v represents ξ we have that $(v, w) = \xi(w) = 0$ for all $w \in W$. We obtain that $\varphi(W^0) \subseteq W^\perp$. If $v \in W^\perp$ then the functional $x \mapsto (v, x)$ cancels over W (by the definition of orthogonality). □

To compute the orthogonal complement of a vector subspace W, we compute $W^\perp = \ker\left(A^T\right)$, where A is the matrix with column vectors in the spanning set of W as its columns. Precisely, W is represented as the column space of A. Proof is available in [9].

References

[1] Filippo Bonchi, Barbara König, and Sebastian Küpper. Up-to techniques for weighted systems (extended version). *CoRR*, abs/1701.05001, 2017.

[2] Filippo Bonchi, Marcello Bonsangue, Michele Boreale, Jan Rutten, and Alexandra Silva. A coalgebraic perspective on linear weighted automata. *Information and Computation*, 211:77 – 105, 2012.

[3] Martin De Wulf, Laurent Doyen, Thomas A Henzinger, and J-F Raskin. Antichains: A new algorithm for checking universality of finite automata. In *International Conference on Computer Aided Verification*, pages 17–30. Springer, 2006.

[4] Chen Fu, Yuxin Deng, David N Jansen, and Lijun Zhang. On equivalence checking of nondeterministic finite automata. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 216–231. Springer, 2017.

[5] Michele Boreale. Weighted bisimulation in linear algebraic form. In *International Conference on Concurrency Theory*, pages 163–177. Springer, 2009.

[6] Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. *ACM SIGPLAN Notices*, 48(1):457–468, 2013.

[7] John E Hopcroft. *A linear algorithm for testing equivalence of finite automata*, volume 114. Defense Technical Information Center, 1971.

[8] Yan-Bin Jia. Singular value decomposition. Available online at `http://web.cs.iastate.edu/~cs577/handouts/svd.pdf`.

[9] Dan Margalit and Joseph Rabinoff. *Interactive Linear Algebra*.