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

September 30, 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, calculates 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 relationship between states. We then compare the two algorithms results to verify their correctness on randomly generated automata samples, and provide some runtime statistics.

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

1 Introduction

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. For ease of implementation, we focus only on weighted automata over the field of real numbers $\mathbb{R}$.

2 Preliminaries and Notation

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, $t = (t_a : X \to K)_{a \in A}$ 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 t_a are represented as $\mathbb{K}^{X \times X}$ matrices. $o \in \mathbb{K}^{1 \times X}$ is represented as a row vector. $t_a(v)$ denotes the vector obtained by multiplying the matrix t_a by the column vector $v \in \mathbb{K}^{X \times 1}$. $o(v)$ denotes the scalar $s \in \mathbb{K}$ obtained by dot product of the row vector o with $v \in \mathbb{K}^{X \times 1}$. A^* is the set of all words over A. ϵ is the empty word and aw is 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, $[\![\cdot]\!] : \mathbb{K}^X \to K^{A^*}$ 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 language equivalent, denoted with $v_1 \sim v_2$ 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.

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.

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

Definition 2.4.

3 Algorithms

3.1 Forward Partition Refinement Algorithm for the Largest Weighted Bisimulation

The algorithm is defined by the iterative method:

$$R_0 = \ker(o)\,, \qquad R_{i+1} = R_i \cap \bigcap_{a \in A} t_a(R_i)^{-1} \tag{1}$$

The algorithm stops when $R_{j+1} = R_j$. Proof is available in section 4.1 of [2]

3.2 Backwards Partition Refinement Algorithm for the Largest Weighted Bisimulation

The algorithm is defined by the iterative method:

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

Where $\ker(o)^0$ is an annihilator. The algorithm stops when $R_{j+1} = R_j$. 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.

See Section **??**.

4.1 Input file format

4.2 Calculating the intersection of linear subspaces

To compute algorithm (1), we need an efficient way to calculate a basis of the intersection of an arbitrary number of linear subspaces of $\mathbb{R}^n$, given their spanning sets.

Definition 4.1. Algorithm for a basis of the intersection of subspaces
Defined in file `lin/intersect.go`. Let's consider two linear subspaces of $\mathbb{R}^n$ defined with their spanning sets of column vectors: $U = \langle u_1, u_2, \ldots, u_p \rangle$ and $W = \langle w_1, w_2, \ldots, w_k \rangle$. We create the block matrix $A \in \mathbb{R}^{n \times (p+k)}$ defined as follows:

$$a_{ij} = \begin{cases} (u_j)_i & \text{for } j = 1, \ldots, p \quad i = 1, \ldots, n \\ (w_{j-p})_i & \text{for } j = p+1, \ldots, p+k \quad i = 1, \ldots, n \end{cases}$$

The matrix will have the form

$$A = \left[\begin{pmatrix} u_1 \end{pmatrix}, \dots \begin{pmatrix} u_p \end{pmatrix}, \begin{pmatrix} w_1 \end{pmatrix}, \dots \begin{pmatrix} w_k \end{pmatrix} \right]$$

It follows that $\ker(A)$ is a basis for $U \cap W$. This can be done with an arbitrary finite number of subspaces of $\mathbb{R}^n$, as long as their spanning set is *finite*.

Definition 4.2. Algorithm for computing the null space of a vector subspace
The algorithm implementation can be found in file `lin/nullspace.go`; it is adapted from [3]. Let's consider the singular value decomposition of a matrix $A \in \mathbb{R}^{m \times n}$:

$$A = U \Sigma V^T \qquad \Sigma = \text{diag}\left(\sigma_1, \sigma_2, \dots, \sigma_{\min(m,n)}\right) \qquad U \in \mathbb{R}^{m \times m} \qquad V \in \mathbb{R}^{n \times n}$$

Where V and U are orthogonal and the singular values are ordered: $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$. It follows that rank (A) is equal to the number of nonzero singular values and a basis of the (right) null space of A is the spanning set of the columns of V corresponding to singular values equal to zero.

Example 4.1. First, we show a shorter Python implementation of the algorithm to compute the nullspace, using the *SciPy* library [3]:

```python
def nullspace(A, atol=1e-13, rtol=0):
    A = np.atleast_2d(A)
    u, s, vh = svd(A)
    tol = max(atol, rtol * s[0])
    nnz = (s >= tol).sum()
    ns = vh[nnz:].conj().T
    return ns
```

The Go implementation is quite longer:

```go
package lin

import (
        "log"
        "math"

        "gonum.org/v1/gonum/mat"
)

const tol = 10e-13

// columns of the returned matrix form an orthonormal basis
// for the nullspace of matrix a, computed
// through svd decomposition. also returns the maximum residual
func Nullspace(a mat.Matrix) (mat.Matrix, float64) {
        // compute svd decomposition
        var svd mat.SVD
        if ok := svd.Factorize(a, mat.SVDFullV); !ok {
                log.Fatal("failed␣to␣factorize␣A")
        }
        vt := mat.NewDense(1, 1, nil)
        vt.Reset()
        svd.VTo(vt)

        // residual
        res := 0.0

        // the (right) null space of A is the columns of vt corresponding to
        // singular values equal to zero.
        j := 0
        for _,  := range svd.Values(nil) {
                if  <= tol {
                        break
                }
                j++
        }

        // compute the residuum
        for k := j; k < vt.RawMatrix().Cols; k++ {
                v := mat.NewVecDense(1, nil)
                v.Reset()
                v.MulVec(a, vt.ColView(k))
                // current residual
                currRes := mat.Norm(v, math.Inf(1))
                if currRes > res {
                        res = currRes
                }
        }

        m, n := vt.Dims()
        ker := vt.Slice(0, m, j, n)
        return ker, res
}
```

4.3 Implementing the backwards algorithm

In [2], for the first step of the iterative backwards algorithm to compute the largest linear weighted bisimulation, we need to compute ker $(o)^0$. 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. *Within finite dimensional spaces, the annihilator W^0 is isomorphic to the orthogonal complement.*

Proof. Let V be an inner product space over the field $\mathbb{K}$ with an inner product $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{K}$ defined. With the inner product defined on V, every linear functional can be represented with a vector: if $\xi : V \to \mathbb{K}$ is a functional, c is an unique vector such that $\forall x \in V \,.\, \langle c, x \rangle = \xi(x)$. The map associating every $\xi \in V^*$ with the vector $c \in V$ that corresponds to ξ with respect to the inner product is a linear isomorphism. Through this isomorphism, the annihilator W^0 becomes $W^\perp$. □

To compute the orthogonal complement of a vector subspace W, we compute $W^\perp = \text{ker}\left(A^T\right)$, 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 [4].

4.3.1 Headings: third level

Paragraph

5 Examples of citations, figures, tables, references

[**?**, **?**] and see [**?**].

The documentation for `natbib` may be found at

`http://mirrors.ctan.org/macros/latex/contrib/natbib/natnotes.pdf`

Of note is the command `\citet`, which produces citations appropriate for use in inline text. For example,

```
   \citet{hasselmo} investigated\dots
```

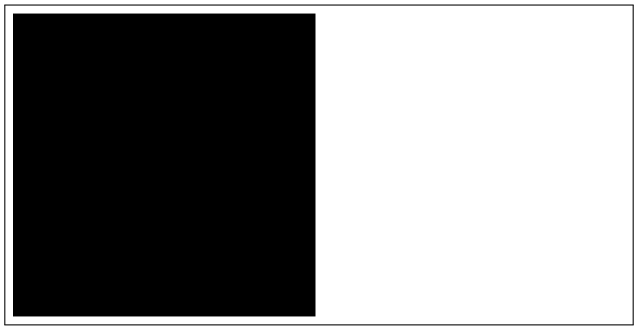

Figure 1: Sample figure caption.

Table 1: Sample table title

Part		
Name	Description	Size (μm)
Dendrite	Input terminal	$\sim$100
Axon	Output terminal	$\sim$10
Soma	Cell body	up to 10^6

produces

Hasselmo, et al. (1995) investigated...

`https://www.ctan.org/pkg/booktabs`

5.1 Figures

See Figure 1. Here is how you add footnotes. [1]

5.2 Tables

See awesome Table 1.

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] Open Source Community. Scipy cookbook: Rank and nullspace of a matrix, 2011. Available online at `https://scipy-cookbook.readthedocs.io/items/RankNullspace.html`.

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

[1] Sample of the first footnote.