DRAFT: GO IMPLEMENTATION OF TECHNIQUES FOR WEIGHTED LANGUAGE EQUIVALENCE

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

October 11, 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 the language equivalence relation in finite state weighted automata (WAs). The first algorithm checks the language equivalence of two vectors (states) in a weighted automaton, with an up-to technique by using an additional data structure representing a congruence relation. The second algorithm, a linear partition refinement algorithm, computes the whole equivalence relation, precisely the largest linear weighted bisimulation, by linearizing the state space and iteratively refining the relation. We then compare results of the two algorithms to verify their correctness and performance on randomly generated samples. We finally provide a comparison and runtime statistics.

1 Introduction

Weighted automata (WAs) are a generalization of non-deterministic automata. When reading a symbol, a non-deterministic automaton can transition in different states simultaneously. Weighted automata introduce *weights* over transitions, which can for example, represent the cost of a transition, or in probabilistic systems, the chance of such transition to happen. WAs can be represented with a set of states, an output function and a set of transition matrices, indexed over the symbols in the alphabet the automaton can read. While those automata are typically defined over semirings, our implementation will focus for simplicity on automata with transitions defined over the real number field. The current configuration of a finite weighted automaton W, defined on n states, will be represented with a column vector of length n, with values over the semiring or field on which transition weights of the automaton W are also defined.

The goal of this work is to provide an high-performance implementation in the Go programming language of two different techniques to compute *weighted language equivalence*. Such equivalence relation is a bisimulation: a relation R is a bisimulation whenever two states v_1, v_2 in R can simulate each other, resulting in a pair that is still in R. Two state vectors v_1, v_2 in a weighted automaton are said to be weighted language equivalent, written as $v_1 \sim_l v_2$, when they simulate each other by accepting the same words with the same resulting output weights.

The first technique we implement, defined in [1], is an up-to technique for weighted language equivalence called HKC. It is defined for weighted systems over arbitrary semirings and can be implemented with set theoretic constructs. The second technique is defined in [2]: a coalgebraic perspective is adopted to define a technique for language equivalence which exploits the linear representation of an automaton. This latter technique "*minimizes*": by linearizing the state space of a weighted automaton, it computes a basis for an entire linear relation (see definition 2.7) which coincides with weighted language equivalence. This technique for finite weighted automata over fields was first introduced by Michele Boreale in [3].

Another example of the comparison between algorithms to compute language equivalence, precisely between HKC and an alternative algorithm called the antichain algorithm ([4]), was published in 2017 [5].

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, $t = \left\{t_a : X \to \mathbb{K}^X\right\}_{a \in A}$ 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 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. *Generating set for the congruence closure:*
Let $\mathbb{K}$ be a field, X a finite set and $R \subseteq \mathbb{K}^X \times \mathbb{K}^X$ a relation. Let $(v, v') \in \mathbb{K}^X \times \mathbb{K}^X$ be a pair of vectors. The generating set is defined as $U_R = \{u - u' \mid (u, u') \in R\}$. Then $(v, v') \in c(R) \iff (v - v') \in U_R$

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 2.5. 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 $t = \left\{t_a \in \mathbb{K}^{V \times V}\right\}_{a \in A}$ 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 seen in [3], 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 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 2.6. The language recognized by a vector $v \in V$ in 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 each symbol 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 the equivalence relation $\approx_{\mathcal{L}}$ for a given LWA $L = (V, o, t)$ as

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

Proofs are 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 2.7. *Linear Relations:*
Let U be a subspace of V. The binary relation R_U over V is defined by

$$v_1 \; R_U \; v_2 \quad \iff \quad v_1 - v_2 \in U$$

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

Definition 2.8. *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

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

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

Definition 2.9. *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 $\ker\left(\llbracket - \rrbracket_V^{\mathcal{L}}\right)$ 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 $\sim_l$:*

3 Algorithms

The first algorithm we implement to compute language equivalence, called `HKC`, is adapted from [1]. The algorithm returns `true` $\iff \llbracket v_1 \rrbracket = \llbracket v_2 \rrbracket$. 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. It is defined for WAs over semirings, but in this implementation 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. 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.

Our implementation is directly modeled on the algorithm shown in [3], since we are fixing weights on $\mathbb{R}$ and computing the orthogonal complements instead of dual spaces and annihilators.

Note. Recall from section 2 that $\approx_{\mathcal{L}}$ is a linear relation, therefore $v_1 \approx_{\mathcal{L}} v_2 \iff (v_1 - v_2) \in \ker(\approx_{\mathcal{L}})$

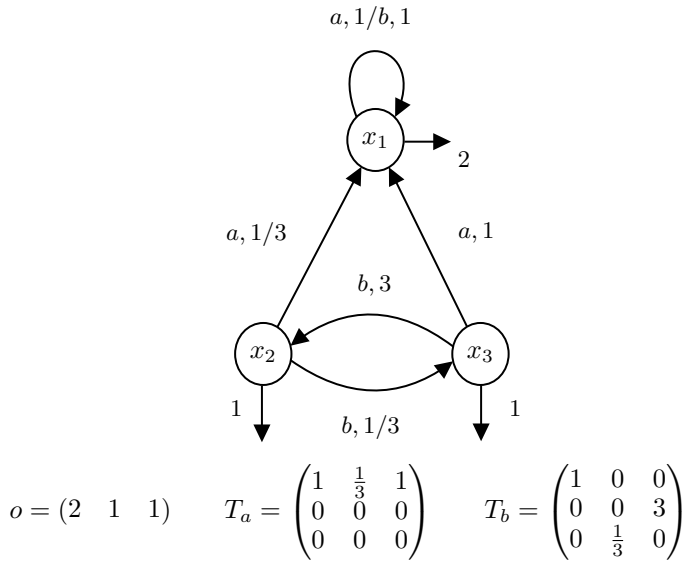

Figure 1: Example of a Weighted Automata

From Lemma 2.3, it follows that given an LWA $L = (V, o, t)$ and a corresponding basis of $\approx_{\mathcal{L}}$ computed by BPR, one can check language equivalence of two vectors in the state space, $v_1 \sim_l v_2$, by checking if $(v_1 - v_2) \in \ker(\approx_{\mathcal{L}})$. Therefore, we can say that BPR "*minimizes*", or it computes the whole binary linear relation $\approx_{\mathcal{L}}$, coinciding with $\sim_l$.

The `BPR` algorithm starts from the basis of 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.

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

In the next sections we will compare execution results of our implementation of the algorithms `BPR` and `HKC` to verify correctness, and to provide insight on runtime results. 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?

3.1 HKC Algorithm

We give the pseudocode definition of the `HKC` procedure from [1]:

Figure 2: The `HKC`(v_1, v_2) procedure

```
1  HKC(v1, v2):
2  R := ∅; todo := ∅
3  while todo is not empty do
4      extract (v1', v2') from todo
5      if (v1', v2') ∈ c(R) then continue
6      if o(v1') ≠ o(v2') then return false
7      for all  a ∈ A
8          insert (ta(v1'), ta(v2')) into todo
9      insert (v1', v2') into R
10 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.

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

The algorithm stops when $R_{j+1} = R_j$. An index $j \leq \dim(V)$ is guaranteed to exist, such that the algorithms stops 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 are implemented in the Go programming language. Real numbers are implemented with double precision floating point numbers, precisely of `float64` type.

This implementation makes use of the Gonum library, an excellent toolkit for high-performance numerical computations. We only import the Gonum libraries for matrices, linear algebra and visual plotting. Although not GPU accelerated, Gonum matrix operations are run on the CPU and accelerated with BLAS and LAPACK.

4.1 Data Structures

In this implementation, the data structure for representing weighted automata is a `struct`:

Figure 3: Source code for the Weighted Automaton data structure, found in `automata/automata.go`

```
1  // This file contains weighted automata data structure definition
2
3  package automata
4
5  import (
6          "gonum.org/v1/gonum/mat"
7  )
8
9  type Automaton struct {
10         // The input alphabet slice
11         A []string
12         // Transition matrices are maps from input strings
13         // to dense real valued matrices
14         T map[string]*mat.Dense
15         // Output vector uses a dense real valued vector
16         O *mat.VecDense
17         // Number of states/dimension of vector space V in LWA
18         Dim int
19         // Col(LLWB) is a basis of the largest linear weighted bisimulation,
20         // a binary linear relation. vRw iff (v-w) in Ker(R)
21         // with LLWBperp, we denote a basis of the orthogonal
22         // complement of the basis LLWB
23         LLWBperp *mat.Dense
24         BPRTol float64 // tolerance value for BPR
25         HKCTol float64 // tolerance value for HKC
26 }
```

Note. Slices in Go are a convenient and efficient extension of the concept of arrays: they provide an abstraction for indexed, variable length sequences of typed data, and provide useful helper functions for creating, appending and selecting elements.

We then provide methods for reading an automaton from a text stream, applying transitions and output functions to vectors, and generating random automata with real and natural valued weights.

4.2 Implementation of HKC

Instead of creating dedicated structs, we exploit the `mat.Dense` data type in Gonum to efficiently represent:

- Sets of vectors with $n \times k$ dense matrices (k is the number of vectors in the set)
- Pairs of vectors with $n \times 2$ dense matrices
- Sets and the "todo" stack of pairs with $n \times 2k$ dense matrices (k is the number of pairs in the set or stack)

To increase efficiency of the methods for inclusion checking and insertion in sets of vectors, one could keep the columns ordered (by vector norm) in the corresponding matrix.
To represent the congruence relation R, we introduce a `struct` containing:

- A dense matrix `s` of size $n \times 2k$ containing the set of pairs in the relation
- A dense matrix `u` to represent the generating set U_R for the congruence closure (see definition 2.4).
- Two integers representing the number of pairs in the set, and the size of vectors.
- A tolerance value to be used in equivalence checks. Best results were obtained with 10^{-14}.

When adding a pair of vectors to the congruence relation, we extend the columns of the matrix `s` with the pair (v_1, v_2), if and only if $(v_1 - v_2)$ is not already in U. To check if the pair is in the congruence closure $c(R)$, we check if $(v_1 - v_2)$ is contained in U.

Figure 4: Implementation of the `HKC`(v_1, v_2) procedure

```
1  // this file contains the implementation of the HKC procedure
2
3  package automata
4
5  import (
6          "math"
7
8          "gonum.org/v1/gonum/mat"
9  )
10
11 // HKC checks the language equivalence of two state vectors for a
12 // given weighted automaton by building a congruence relation
13 func (a Automaton) HKC(v1, v2 *mat.VecDense) (bool, error) {
14         rel := NewRelation(a.HKCTol, a.Dim)
15         todo := NewPairStack()
16
17         p, err := NewPair(v1, v2)
18         if err != nil {
19                 return false, err
20         }
21
22         // insert (v1, v2) into the todo list
23         todo = PairStackPush(todo, p)
24
25         for !todo.IsEmpty() {
26                 // extract (v1', v2') from todo
27                 q, err := PairStackPop(todo)
28                 if err != nil {
29                         return false, err
30                 }
31
32                 if rel.PairIsInCongruenceClosure(q) {
33                         continue
34                 }
35
36                 o1 := a.GetOutput(PairLeft(q))
37                 o2 := a.GetOutput(PairRight(q))
38                 if math.Abs(o1-o2) > a.HKCTol {
39                         return false, nil
40                 }
41
42                 for _, sym := range a.A {
```

```
43
44                      w1 := a.ApplyTransition(sym, PairLeft(q))
45                      w2 := a.ApplyTransition(sym, PairRight(q))
46                      wp, err := NewPair(w1, w2)
47                      if err != nil {
48                              return false, err
49                      }
50
51                      PairStackPush(todo, wp)
52              }
53
54              // insert (v1', v2') into R
55              rel.Add(q)
56      }
57
58      return true, nil
59  }
```

4.3 Implementation of BPR

Given a WA L, at the first step of BPR, we set $R_0 = o$, with o being the dense vector representing the output function of L, as seen in [2]. To compute R_{i+1} at each step, the implemented BPR algorithm:

1. Computes $t_a^T(R_i)$ through matrix multiplication for each $a \in A$
2. Concatenates the resulting matrices to R_i in a resulting matrix G
3. Computes R_{i+1} as the orthonormal basis of the column space of G, through singular value decomposition.

At the end of BPR, we store the basis for the orthogonal complement of $\approx_{\mathcal{L}}$ as an attribute of the automaton. To check if two vectors $v_1 \approx_{\mathcal{L}} v_2$, we check that $R_j^T(v_1 - v_2) = \vec{0}$, with a prefixed tolerance.
To compute a basis for $\approx_{\mathcal{L}}$, at the last step of the algorithm, we would 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 annhilator 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 whose column space is W.

Figure 5: Implementation of Backwards Partition Refinement

```
1  // this file contains definitions for the backwards algorithm for
2  // computing the largest linear weighted bisimulation
3
4  package automata
5
6  import (
7          "log"
8
9          "github.com/0x0f0f0f/lwa-techniques/lin"
10         "gonum.org/v1/gonum/mat"
11 )
12
13 // BackwardsPartitionRefinement computes and stores a basis for the
```

```
14 // complement of the largest linear weighted bisimulation of
15 // the linear weighted automaton. returns the condition number
16 func (a *Automaton) BackwardsPartitionRefinement() float64 {
17         // i = 0
18         lastBasis := mat.NewDense(a.Dim, 1, a.O.RawVector().Data)
19         currBasis := lastBasis
20         // condition number
21         lastCond := 0.0
22
23         for i := 1; i <= a.Dim; i++ {
24                 // \sum_{a \in A} T_a^T(R_i)
25                 for _, sym := range a.A {
26                         newBasis := a.ApplyTransposeTransitionBasis(sym, lastBasis)
27                         currBasis = lin.Union(currBasis, newBasis)
28                 }
29                 tmp, cond := lin.OrthonormalColumnSpaceBasis(currBasis, a.BPRTol)
30                 currBasis = tmp.(*mat.Dense)
31                 lastBasis = currBasis
32                 lastCond = cond
33         }
34
35         a.LLWBperp = currBasis
36         // we could compute the orthogonal complement to find a basis of LLWB:
37         // a.LLWB = lin.Complement(currBasis).(*mat.Dense)
38         return lastCond
39 }
40
41 // BPREquivalence checks the equivalence of 2 vectors
42 // after a basis of the LLWB is computed through BPR,
43 func (a Automaton) BPREquivalence(v1, v2 *mat.VecDense) bool {
44         if a.LLWBperp == nil {
45                 log.Fatalln("largest linear weighted bisimulation not computed for
                      automaton")
46                 return false
47         }
48
49         sub := mat.VecDenseCopyOf(v1)
50         sub.SubVec(v1, v2)
51
52         mul := mat.VecDenseCopyOf(sub)
53         mul.Reset()
54         mul.MulVec(a.LLWBperp.T(), sub)
55
56         return lin.IsZeroTol(mul, a.BPRTol)
57 }
```

Note. **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 $\text{rank}\,(A)$ is equal to the number of nonzero singular values, and as explained in [8]:

1. $\text{rank}\,(A) = \text{rank}\,(\Sigma) = r$
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 column space. The implementation can be found in files `lin/colspace.go` and `lin/nullspace.go`.

5 Runtime Results

We the studied of the influence of the tolerance value over the correctness of the algorithms results. Tests took place on randomly generated automata. A basis of $\approx_{\mathcal{L}}$ was computed for each automaton with BPR. An arbitrary number of vector pairs were generated as an uniformly distributed random linear combination of the vectors in the spanning set of $\approx_{\mathcal{L}}$. The same quantity of vector pairs was generated randomly with an uniform distribution. BPREquivalence and HKC procedures were then executed with those pairs in input.
By defining pairs of vectors as

- *true positives* (TP) if reported as language equivalent by both procedures
- *true negatives* (TN) if reported as not language equivalent by both procedures
- *false positives* (FP) if reported as language equivalent by HKC but not by BPR
- *false negatives* (FN) if reported as language equivalent by BPR but not by HKC

We have introduced *accuracy, precision, recall* and *F1 score* over the tests results:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{F1} = 2 * \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

We have then collected the F1 score in relation to varying tolerance values. The plot is shown in Figure 6. The program was executed on an x86_64 AMD Ryzen 5 2600X Six Core CPU, running at 3.6GHz with 32GB RAM, running Void Linux. Executed sequentially, F1 score tests took 2m26.19s Parallelized with a worker pool, the tests took 1m13.12s.
TODO: BPR conditioning
TODO empty LLWB chance, memory and time plots, varying automata size parameters

6 Conclusion and Future Work

During the testing phase, we have found that on randomly generated automata over the $\mathbb{R}$ field, the probability of BPR returning an empty basis of $\approx_{\mathcal{L}}$ increases together with the number of states in the automaton, and the number of symbols in the alphabet. In a given linear weighted automaton $L = (V, o, t)$ over the $\mathbb{R}$ field and an alphabet A, the chance of BPR returning an empty basis also increases together with the maximum weight in modulo of the transition matrices, $\forall a \in A,\ \max |(t_a)_{ij}|$, for $i, j = 1 \ldots, n$, with $n = \dim(V)$. Authors of [2] have confirmed that it is normal for this to happen and that this fact is not due to an implementation error.
Since the tests in this work rely heavily on a non-empty basis of $\approx_{\mathcal{L}}$ to generate language equivalent vectors, the automata with an empty $\ker(\approx_{\mathcal{L}})$ had to be ignored during the tests. This introduced substantial overhead as the chance of BPR returning an empty basis grew closer to 1, together with the various parameter of the automata: most of the computing time was spent on generating and running BPR on automata which did not contain any language equivalent vectors, making tests on large automata not possible.
To provide better runtime results for the language equivalence problem, a topic worth further attention is the development of a semi-randomized method to generate structured large sized automata that have a low chance of BPR returning an empty basis of $\approx_{\mathcal{L}}$.
Another idea for the future of this implementation, is the addition of a `Semiring` interface type that would permit much more insightful analysis on weighted automata, not only on the field of real numbers.
To further verify of the algorithms implemented in this work, and any additional algorithm that may be implemented, one could compare the results of this package with the results of the PAWS tool for the analysis of weighted systems [9], developed at the University of Duisburg-Essen.

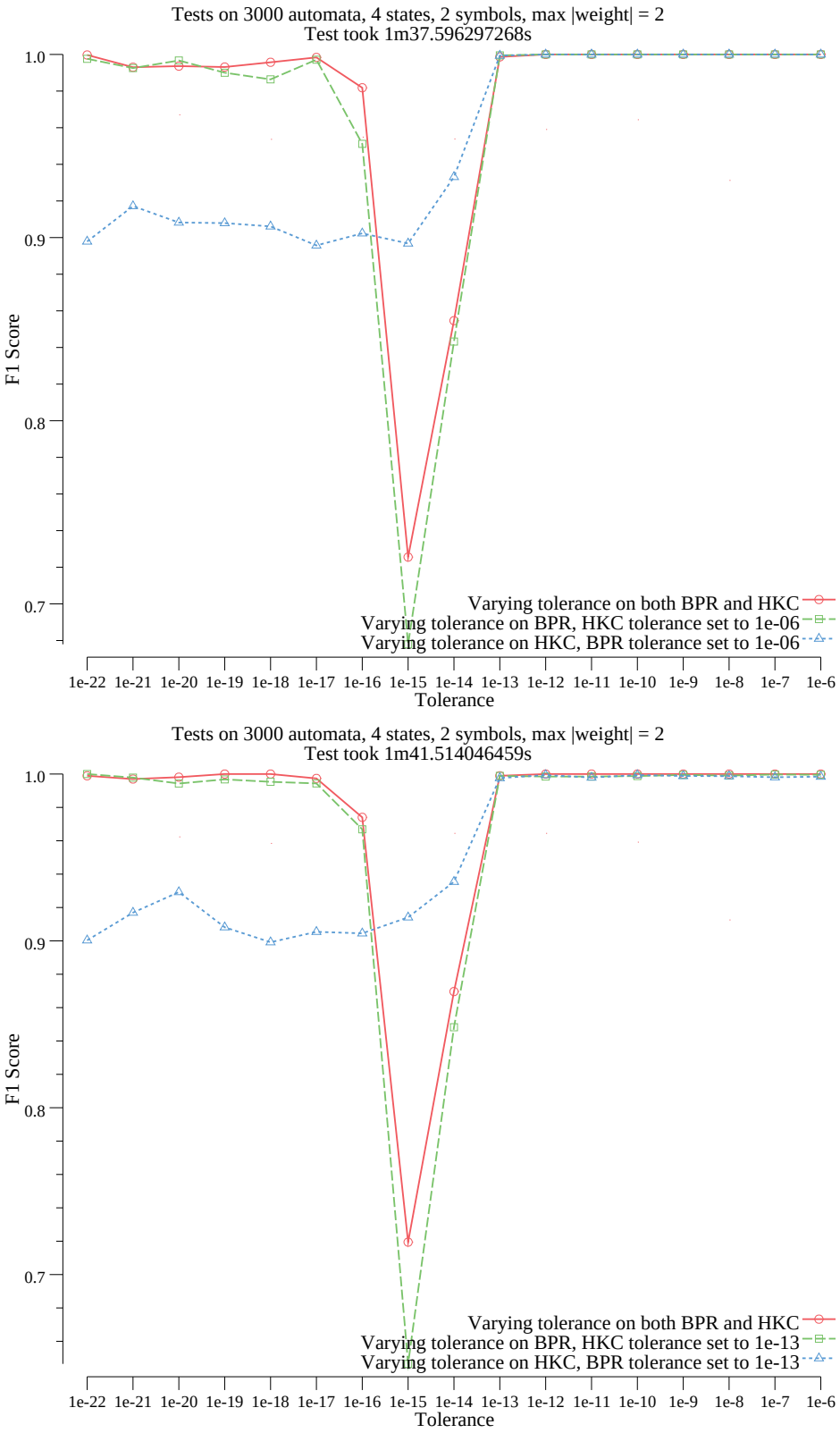

Figure 6: F1 score over tolerance tests, 2000 vector pairs tested per automaton.

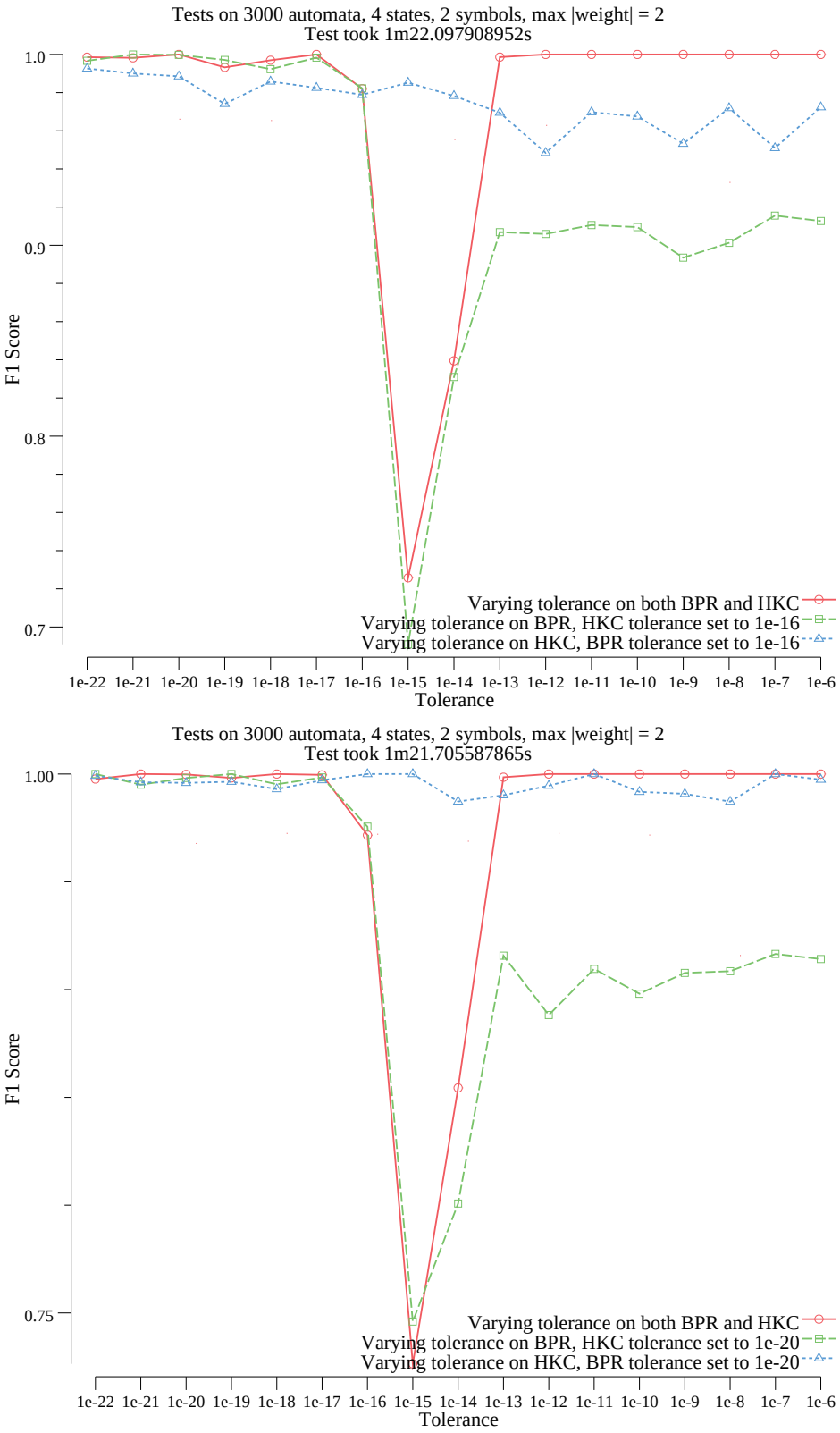

Figure 7: F1 score over tolerance tests, 2000 vector pairs tested per automaton.

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

[4] 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.

[5] 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.

[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] Barbara König, Sebastian Küpper, and Christina Mika. Paws: A tool for the analysis of weighted systems. *arXiv preprint arXiv:1707.04125*, 2017.