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, for simplicity, our implementation will focus only on automata with transitions defined over the field of real numbers. 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, and $\dim(V) = n$; We have that $o : V \to \mathbb{K}$ is a linear map associating to each state its output weight, and $t = \{t_a \in \mathbb{K}^{n \times n}\}_{a \in A}$ 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 $\dim(L) = \dim(V) = n$.

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 $[\![v]\!]^{\mathcal{L}}_V(w) = o(v_n)$ where v_n is the vector reached from v through the composition of the transition functions corresponding to each symbol in w.

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

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

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

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

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

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 $\ker\left([\![-]\!]_V^{\mathcal{L}}\right)$ 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 $\sim_l$:*

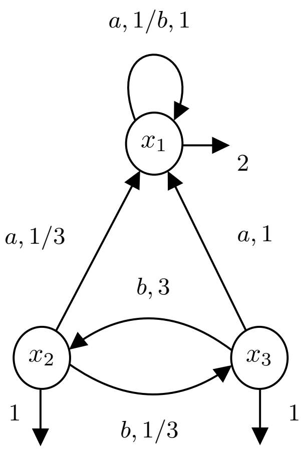
3 Algorithms

The first algorithm we implement to compute language equivalence, called `HKC`, is adapted from [1]. The algorithm returns `true` $\iff$ $[\![v_1]\!] = [\![v_2]\!]$. 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 $v_1 \approx_{\mathcal{L}} v_2 \iff (v_1 - v_2) \in \ker(\approx_{\mathcal{L}})$



$$o = \begin{pmatrix} 2 & 1 & 1 \end{pmatrix} \qquad T_a = \begin{pmatrix} 1 & \frac{1}{3} & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad T_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & \frac{1}{3} & 0 \end{pmatrix}$$

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

```
HKC(v1, v2):
R := ∅; todo := ∅
insert (v1, v2) into 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.

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`

```go
// This file contains weighted automata data structure definitions

package automata

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

// Automaton represents a general finite weighted automaton on the
// real number field
type Automaton struct {
	// The input alphabet slice
	A []string
	// Transition matrices are maps from input strings
	// to dense real valued matrices
	T map[string]*mat.Dense
	// Output vector uses a dense real valued vector
	O *mat.VecDense
	// Number of states/dimension of vector space V in LWA
	Dim int
	// Col(LLWB) is a basis of the largest linear weighted bisimulation,
	// a binary linear relation. vRw iff (v-w) in Ker(R)
	// with LLWBperp, we denote a basis of the orthogonal
	// complement of the basis LLWB
	LLWBperp *mat.Dense
	BPRTol float64 // tolerance value for BPR
	HKCTol float64 // tolerance value for HKC
}
```

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.

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

```go
// this file contains the implementation of the HKC procedure

package automata

import (
	"math"

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

// HKC checks the language equivalence of two state vectors for a
// given weighted automaton by building a congruence relation
func (a Automaton) HKC(v1, v2 *mat.VecDense) (bool, error) {
	rel := NewRelation(a.HKCTol, a.Dim)
	todo := NewPairStack()

	p, err := NewPair(v1, v2)
	if err != nil {
		return false, err
	}

	// insert (v1, v2) into the todo list
	todo = PairStackPush(todo, p)

	for !todo.IsEmpty() {
		// extract (v1’, v2’) from todo
		q, err := PairStackPop(todo)
		if err != nil {
			return false, err
		}

		if rel.PairIsInCongruenceClosure(q) {
			continue
		}

		o1 := a.GetOutput(PairLeft(q))
		o2 := a.GetOutput(PairRight(q))
		if math.Abs(o1-o2) > a.HKCTol {
			return false, nil
```

```
		}

		for _, sym := range a.A {

			w1 := a.ApplyTransition(sym, PairLeft(q))
			w2 := a.ApplyTransition(sym, PairRight(q))
			wp, err := NewPair(w1, w2)
			if err != nil {
				return false, err
			}

			PairStackPush(todo, wp)
		}

		// insert (v1', v2') into R
		rel.Add(q)
	}

	return true, nil
}
```

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 R_j, 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 given 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 $\langle v, w \rangle = \xi(w) = 0$ for all $w \in W$. We obtain that $\varphi(W^0) \subseteq W^\perp$. If $v \in W^\perp$ then the functional $\xi \mapsto \langle v, x \rangle$ 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

```
// this file contains definitions for the backwards algorithm for
// computing the largest linear weighted bisimulation

package automata

import (
	"log"

	"github.com/0x0f0f0f/lwa-techniques/lin"
	"gonum.org/v1/gonum/mat"
```

```go
)

// BackwardsPartitionRefinement computes and stores a basis for the
// complement of the largest linear weighted bisimulation of
// the linear weighted automaton. returns the condition number
func (a *Automaton) BackwardsPartitionRefinement() float64 {
	// i = 0
	lastBasis := mat.NewDense(a.Dim, 1, a.O.RawVector().Data)
	currBasis := lastBasis
	// condition number
	lastCond := 0.0

	for i := 1; i <= a.Dim; i++ {
		// \sum_{a \in A} T_a^T(R_i)
		for _, sym := range a.A {
			newBasis := a.ApplyTransposeTransitionBasis(sym, lastBasis)
			currBasis = lin.Union(currBasis, newBasis)
		}
		tmp, cond := lin.OrthonormalColumnSpaceBasis(currBasis, a.BPRTol)
		currBasis = tmp.(*mat.Dense)
		lastBasis = currBasis
		lastCond = cond
	}

	a.LLWBperp = currBasis
	// we could compute the orthogonal complement to find a basis of LLWB:
	// a.LLWB = lin.Complement(currBasis).(*mat.Dense)
	return lastCond
}

// BPREquivalence checks the equivalence of 2 vectors
// after a basis of the LLWB is computed through BPR,
func (a Automaton) BPREquivalence(v1, v2 *mat.VecDense) bool {
	if a.LLWBperp == nil {
		log.Fatalln("largest linear weighted bisimulation not computed for
		    automaton")
		return false
	}

	sub := mat.VecDenseCopyOf(v1)
	sub.SubVec(v1, v2)

	mul := mat.VecDenseCopyOf(sub)
	mul.Reset()
	mul.MulVec(a.LLWBperp.T(), sub)

	return lin.IsZeroTol(mul, a.BPRTol)
}
```

Note. **Applications of SVD**

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

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

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 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 uniformly distributed random linear combinations 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 then borrow four concepts from binary classification: *accuracy, precision, recall* and *F1 score*:

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

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

We have then computed the F1 score in relation to varying tolerance values. The plot is shown in Figures 6 and 7. 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 an average 2m26.19s, Parallelized by using a worker pool, the tests lasted the times reported in Figure 6 and 7. For example, in the first plot of Figure 7, the test took 82 seconds. Since we computed BPR for 3000 random automata, over 17 tolerance values, for 3 line plots, with a chance of finding a non-empty basis for $\approx_{\mathcal{L}}$ of approximately $1/3$, the BPR algorithm was executed an approximate average of $\frac{3000 \cdot 3 \cdot 17}{82} \approx 1865$ times per second. The HKC algorithm, was executed 2000 times on each automaton with a non-empty basis of $\approx_{\mathcal{L}}$, therefore, it was executed an approximate average of $\frac{3000 \cdot 17 \cdot 2000 \cdot 3}{3 \cdot 82} \approx 1243902$ times per second.

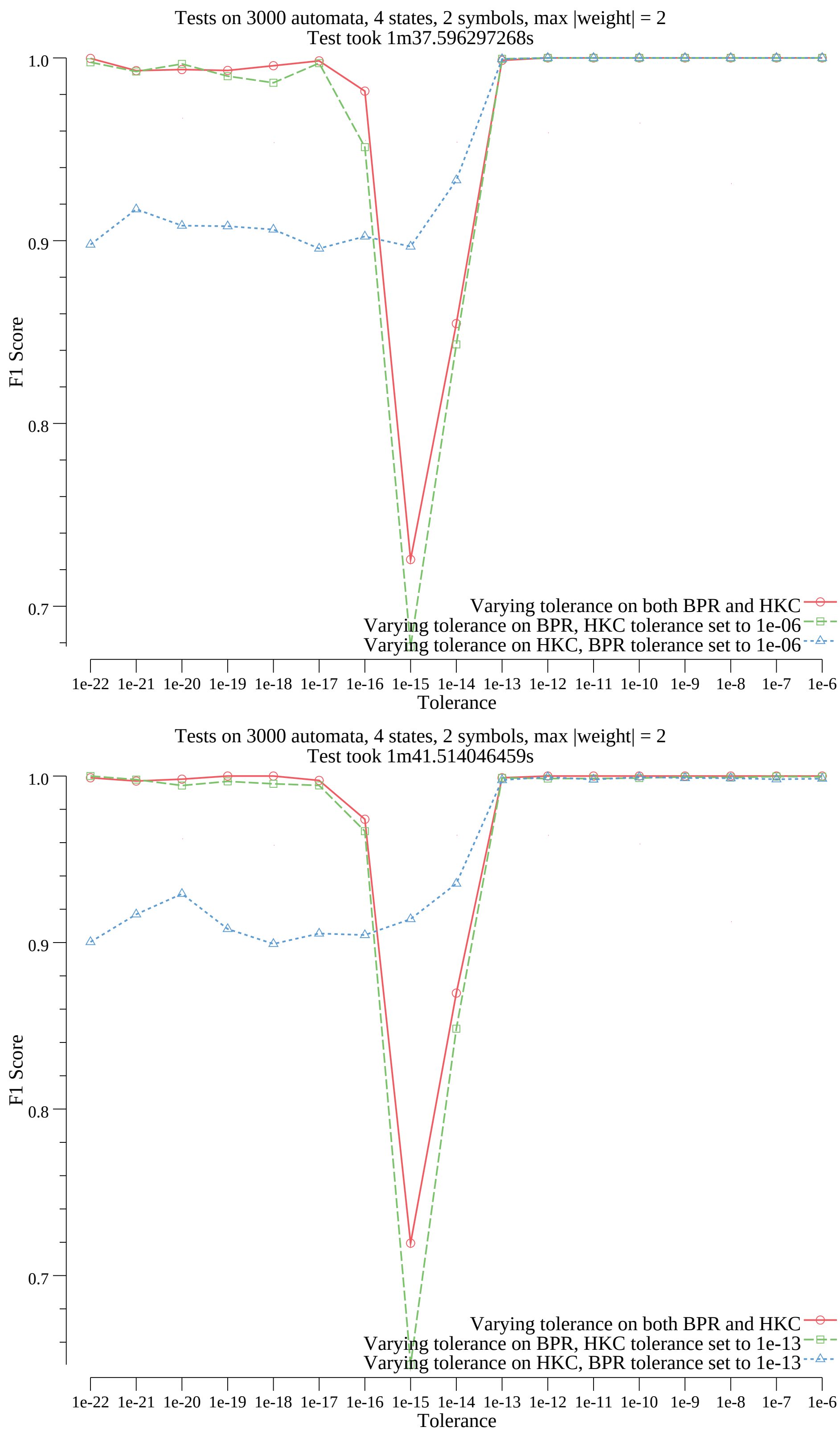
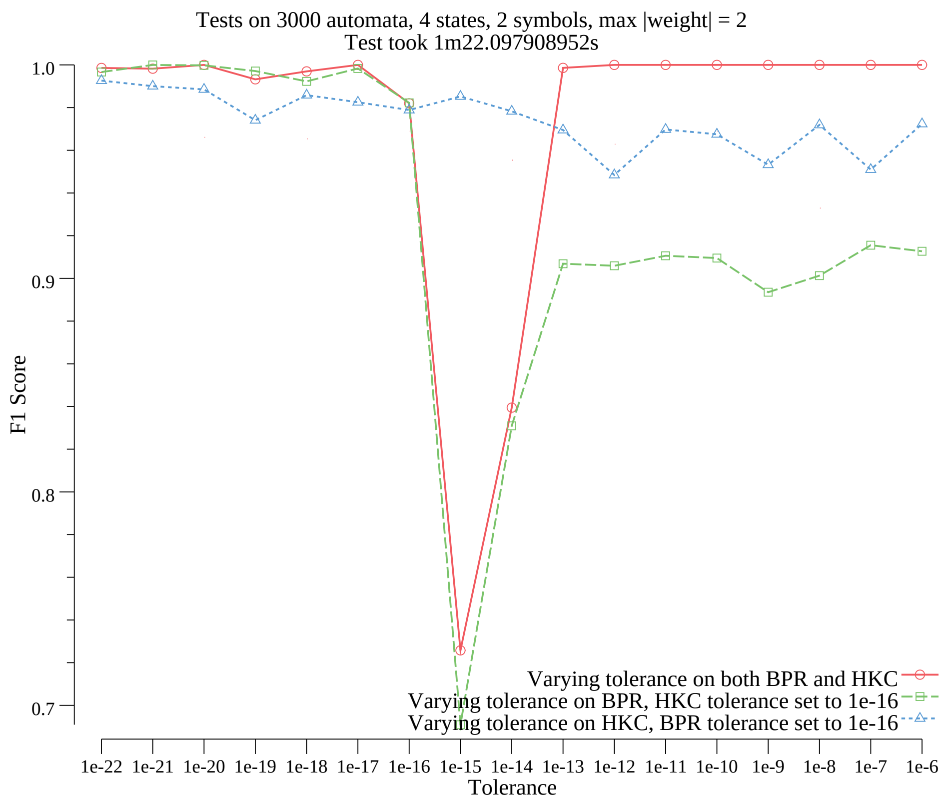


Figure 6: F1 score over tolerance tests, 2000 vector pairs tested per automaton.



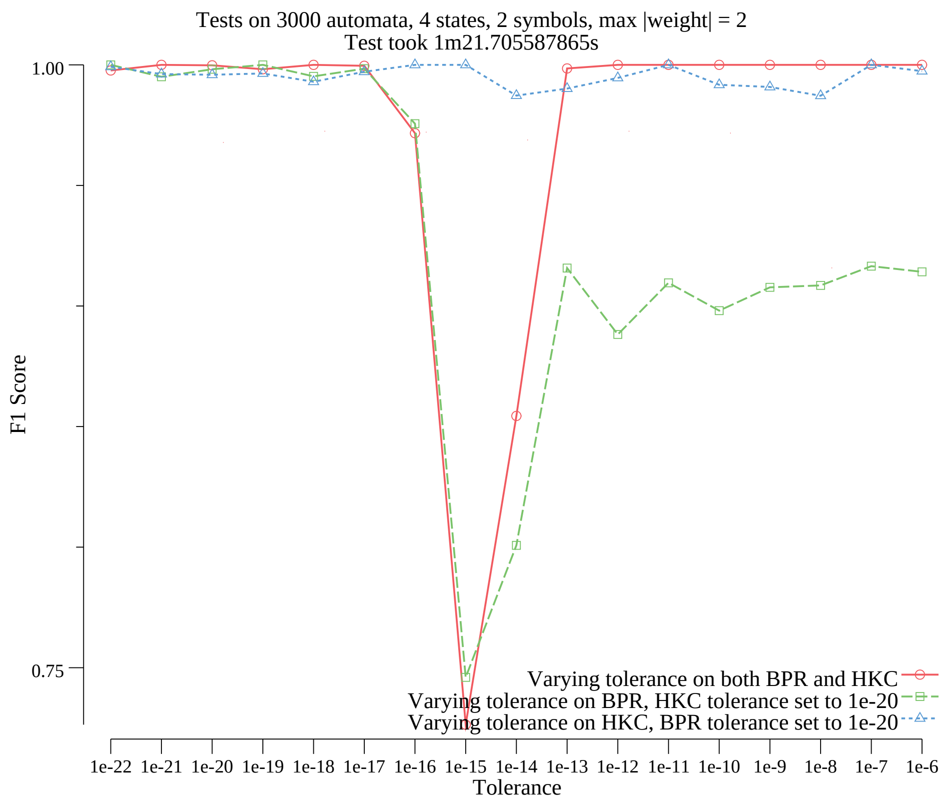


Figure 7: F1 score over tolerance tests, 2000 vector pairs tested per automaton.

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 topic worth further investigation is the cause of the sudden drop at 1e-15 in the F1 score plots.
An 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.

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.

A Complete Code

A.1 Linear Algebra Utilities

Figure 8: `lin/util.go`

```
// this file contains miscellaneous utility functions for various
// linear algebra applications using Gonum

package lin

import (
```

```go
	"errors"
	"fmt"
	"math"
	"math/rand"

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

// IsZero returns true if a vector is composed only of zero values
func IsZero(vec *mat.VecDense) bool {
	for _, el := range vec.RawVector().Data {
		if el != 0 {
			return false
		}
	}

	return true
}

// IsZeroTol returns true if a vector is composed only of zero values, with a tolerance
    of tol
func IsZeroTol(vec *mat.VecDense, tol float64) bool {
	for _, el := range vec.RawVector().Data {
		if math.Abs(el) > tol {
			return false
		}
	}

	return true
}

// EyeDense creates an n*n identity matrix
func EyeDense(n int) *mat.Dense {
	a := mat.NewDense(n, n, nil)
	for i := 0; i < n; i++ {
		a.Set(i, i, 0)
	}
	return a
}

// generate a random float64 between -w and w
func randFloat64(w float64) float64 {
	sign := 1.0
	if rand.Intn(2) == 0 {
		sign = -1.0
	}
	return sign * rand.Float64() * w
}

// RandDense generates a normally distributed random n*n matrix
func RandDense(n int, maxweight float64) *mat.Dense {
	data := make([]float64, n*n)
	for i := range data {
		data[i] = randFloat64(maxweight)
	}
	return mat.NewDense(n, n, data)
}

// RandIntDense generates a normally distributed random integer n*n matrix
```

```
func RandIntDense(n, max int) *mat.Dense {
	data := make([]float64, n*n)
	for i := range data {
		data[i] = float64(rand.Intn(max))
	}
	return mat.NewDense(n, n, data)
}

// PokeHoles sets z randomly chosen values in the matrix to 0
func PokeHoles(a *mat.Dense, z int) {
	m, n := a.Dims()
	for k := 0; k < z; k++ {
		i := rand.Intn(m)
		j := rand.Intn(n)

		a.Set(i, j, 0)
	}
}

// LinearCombination performs a linear combination of the columns of v and the coefficients
// in the elements of b
func LinearCombination(a *mat.Dense, b *mat.VecDense) *mat.VecDense {
	m, n := a.Dims()
	bn, _ := b.Dims()

	if n != bn {
		panic(errors.New("mat-vector dimension mismatch"))
	}

	res := mat.NewVecDense(m, nil)

	for i := 0; i < n; i++ {
		res.AddScaledVec(res, b.AtVec(i), a.ColView(i))
	}

	return res
}

// RandVec generate a normally distributed random n*1 vector
func RandVec(n int, maxweight float64) *mat.VecDense {
	data := make([]float64, n)
	for i := range data {
		data[i] = randFloat64(maxweight)
	}
	return mat.NewVecDense(n, data)
}

// RandNatVec generate a normally distributed random n*1 vector on natural numebrs
func RandNatVec(n, max int) *mat.VecDense {
	data := make([]float64, n)
	for i := range data {
		data[i] = float64(rand.Intn(max))
	}
	return mat.NewVecDense(n, data)
}

// ZeroDense generates a zero matrix of size m*n
func ZeroDense(m, n int) *mat.Dense {
```

```
        return mat.NewDense(m, n, nil)
}

// ZeroVec generates a zero vector of length n
func ZeroVec(n int) *mat.VecDense {
        return mat.NewVecDense(n, nil)
}

// StringMat returns a string representation of a matrix
func StringMat(a mat.Matrix) string {
        return fmt.Sprintf("%.5g", mat.Formatted(a, mat.Squeeze()))
}

// PrintMat prints a matrix to stdout
func PrintMat(a mat.Matrix) {
        fmt.Println(StringMat(a))
}

// CleanTolDense sets elements of a to 0 if they are < tol (in abs)
func CleanTolDense(a *mat.Dense, tol float64) {
        m, n := a.Dims()
        for i := 0; i < m; i++ {
                for j := 0; j < n; j++ {
                        if math.Abs(a.At(i, j)) < tol {
                                a.Set(i, j, 0)
                        }
                }
        }
}
```

Figure 9: `lin/subspace.go`

```
package lin

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

// InSubspace returns true iff the vector b is included in U's column space
// to do so we check if the matrix U and [U|b] have the same rank
func InSubspace(u *mat.Dense, b *mat.VecDense, tol float64) bool {
        var svd mat.SVD
        svd.Factorize(u, mat.SVDNone)

        ranku := svd.Rank(tol)

        // we use the same backing data for the matrix U and [U|b]
        // we then check the rank with SVD factorization
        r, c := u.Dims()
        u = u.Grow(0, 1).(*mat.Dense)
        u.SetCol(c, b.RawVector().Data)
        svd.Factorize(u, mat.SVDNone)
        rankub := svd.Rank(tol)
        u = u.Slice(0, r, 0, c).(*mat.Dense)

        return ranku == rankub
}

// Intersect computes a basis for the intersection of many vector spaces
```

```go
func Intersect(tol float64, spansets ...mat.Matrix) mat.Matrix {
	a := mat.DenseCopyOf(spansets[0])
	for i, span := range spansets {
		if i == 0 {
			continue
		}

		ar, ac := a.Dims()
		_, sc := span.Dims()
		m := mat.NewDense(ar, ac+sc, nil)
		m.Augment(a, span)
		a = m
	}
	// compute the nullspace
	ker, _ := Nullspace(a, tol)
	return ker
}

// Union computes a basis for the union of many vector spaces
func Union(spansets ...*mat.Dense) *mat.Dense {
	n, j := spansets[0].Dims()

	basis := mat.DenseCopyOf(spansets[0])
	for k := 1; k < len(spansets); k++ {
		r, c := spansets[k].Dims()
		if r != n {
			panic("mat dimension mismatch")
		}
		newbasis := mat.NewDense(n, j+c, nil)
		newbasis.Augment(basis, spansets[k])
		j = j + c
		basis = newbasis
	}

	return basis
}

// Complement computes the orthogonal complement of a vector subspace of R^n
func Complement(spanset mat.Matrix, tol float64) mat.Matrix {
	n, m := spanset.Dims()
	// the orthogonal complement of {} is R^n
	if m == 0 {
		return EyeDense(n)
	}

	basis, _ := Nullspace(spanset.T(), tol)
	return basis
}
```

Figure 10: lin/nullspace.go

```go
package lin

import (
	"log"
	"math"

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

```go

// Nullspace computes a null space basis of a given matrix,
// with respect to tolerance.
// 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, tol float64) (mat.Matrix, float64) {
	// compute svd decomposition O(n^3)
	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()
	if n == j {
		return ZeroDense(m, 1), 0
	}
	ker := vt.Slice(0, m, j, n)
	return ker, res
}
```

Figure 11: lin/colspace.go

```go
package lin

import (
	"log"
	"math"

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

```

// OrthonormalColumnSpaceBasis computes an orthonormal basis of the column space of a through
// svd decomposition. Cost is O(n^3). Also returns the condition number
func OrthonormalColumnSpaceBasis(a mat.Matrix, tol float64) (mat.Matrix, float64) {
	var svd mat.SVD
	if ok := svd.Factorize(a, mat.SVDFullU); !ok {
		log.Fatal("failed to factorize A")
	}
	u := mat.NewDense(1, 1, nil)
	u.Reset()
	svd.UTo(u)
	//fmt.Println(mat.Cond(a, 2))
	//fmt.Println(mat.Cond(u, 2))

	// The column space of A is spanned by the first r columns of U.
	j := 0
	for _, := range svd.Values(nil) {
		if math.Abs() <= tol {
			break
		}
		j++
	}

	m, _ := u.Dims()
	//fmt.Println(m, n, j)
	basis := u.Slice(0, m, 0, j)
	return basis, svd.Cond()
}
```

A.2 Automata Data Structures and Methods

Figure 12: automata/automata.go

```
// This file contains weighted automata data structure definitions

package automata

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

// Automaton represents a general finite weighted automaton on the
// real number field
type Automaton struct {
	// The input alphabet slice
	A []string
	// Transition matrices are maps from input strings
	// to dense real valued matrices
	T map[string]*mat.Dense
	// Output vector uses a dense real valued vector
	O *mat.VecDense
	// Number of states/dimension of vector space V in LWA
	Dim int
	// Col(LLWB) is a basis of the largest linear weighted bisimulation,
	// a binary linear relation. vRw iff (v-w) in Ker(R)
	// with LLWBperp, we denote a basis of the orthogonal
	// complement of the basis LLWB
	LLWBperp *mat.Dense
```

```
	BPRTol float64 // tolerance value for BPR
	HKCTol float64 // tolerance value for HKC
}
```

Figure 13: automata/methods.go

```
// this file contains transition functions methods.

package automata

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

// GetOutput applies the output function to a given state vector (o * v)
func (a Automaton) GetOutput(v *mat.VecDense) float64 {
	res := mat.Dot(a.O, v)
	return res
}

// ApplyTransition applies (multiplies) a transition function for a given symbol s
// to a vector v
func (a Automaton) ApplyTransition(s string, v *mat.VecDense) *mat.VecDense {
	res := mat.VecDenseCopyOf(v)
	res.MulVec(a.T[s], v)
	return res
}

// ApplyTransposeTransition applies
// (multiplies) a transpose transition function for a given symbol s, to a vector v
func (a Automaton) ApplyTransposeTransition(s string, v *mat.VecDense) *mat.VecDense {
	res := mat.VecDenseCopyOf(v)
	res.MulVec(a.T[s].T(), v)
	return res
}

// ApplyTransposeTransitionBasis applies
// (multiplies) a transpose transition function for a given symbol s, to a
// matrix b, which column space spans a subspace
func (a Automaton) ApplyTransposeTransitionBasis(s string, b *mat.Dense) *mat.Dense {
	res := mat.DenseCopyOf(b)
	res.Mul(a.T[s].T(), b)
	return res
}
```

Figure 14: automata/io.go

```
package automata

import (
	"bufio"
	"bytes"
	"errors"
	"fmt"
	"io"
	"os"
	"sort"
	"strconv"
	"strings"
```

```

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

func errRead(msg string) error { return errors.New("could not read automaton:" + msg) }

// Helper function to sort and deduplicate in-place a slice of strings
func dedupStr(in []string) []string {
	sort.Strings(in)

	j := 0
	for i := 1; i < len(in); i++ {
		if in[j] == in[i] {
			continue
		}
		j++
		in[j] = in[i]
	}

	return in[:j+1]
}

// Read a positive number
func readIntPos(scanner *bufio.Scanner) (n int, err error) {
	// Read the number of rows
	if !scanner.Scan() {
		err = errRead("found eof when reading number")
		return
	}
	n, err = strconv.Atoi(scanner.Text())
	if err != nil {
		return
	}
	if n <= 0 {
		err = errRead(fmt.Sprintf("number %d must be positive", n))
	}
	return
}

// Read a slice of floats positive number
func readFloat64Slice(scanner *bufio.Scanner) ([]float64, error) {
	if !scanner.Scan() {
		return nil, errRead("found eof when reading vector data")
	}
	fields := strings.Fields(scanner.Text())
	data := make([]float64, 0)
	for _, str := range fields {
		num, err := strconv.ParseFloat(str, 64)
		if err != nil {
			return nil, errRead("could not read float64 value")
		}
		data = append(data, num)
	}
	return data, nil
}

// Helper function to read a dense float matrix
func readDense(scanner *bufio.Scanner, rows, cols int) (*mat.Dense, error) {
	data := make([]float64, 0)
```

```go
	for i := 0; i < rows; i++ {
		// Read a matrix line
		row, err := readFloat64Slice(scanner)
		if err != nil {
			return nil, err
		}
		if len(row) != cols {
			return nil, errRead(fmt.Sprintf("number of elements read in line %d does not match number of columns", i))
		}
		// Append the row to the matrix
		data = append(data, row...)
	}
	return mat.NewDense(rows, cols, data), nil
}

// ReadAutomaton reads an linear weighted automaton from
// a reader. If prompt is true
// then questions are asked (printed to stderr) before scanning
func ReadAutomaton(r io.Reader, prompt bool) (*Automaton, error) {
	a := &Automaton{}
	scanner := bufio.NewScanner(r)
	scanner.Split(bufio.ScanLines)

	// Read the first line containing the alphabet
	if prompt {
		fmt.Fprintf(os.Stderr, "enter the automaton alphabet:\n")
	}
	if !scanner.Scan() {
		return nil, errRead("could not read alphabet")
	}
	a.A = strings.Fields(scanner.Text())
	// Deduplicate elements in the alphabet
	a.A = dedupStr(a.A)
	if len(a.A) < 1 {
		return nil, errRead("alphabet is empty")
	}

	// Read the number of states
	if prompt {
		fmt.Fprintf(os.Stderr, "enter the number of states:\n")
	}
	numStates, err := readIntPos(scanner)
	if err != nil {
		return nil, err
	}
	a.Dim = numStates

	// Read the output vector
	if prompt {
		fmt.Fprintf(os.Stderr, "enter the output weight vector:\n")
	}
	outv, err := readFloat64Slice(scanner)
	if err != nil {
		return nil, err
	}
	if len(outv) != numStates {
		return nil, errRead("output vector length must be equal to the number of states")
```

```go
	}
	a.O = mat.NewVecDense(len(outv), outv)

	fmt.Println(a.O.T().Dims())

	// Read a transition matrix for each symbol in the alphabet
	a.T = make(map[string]*mat.Dense)
	for _, symb := range a.A {
		if prompt {
			fmt.Fprintf(os.Stderr, "enter a %dx%d matrix:\n", numStates,
			    numStates)
		}
		m, err := readDense(scanner, numStates, numStates)
		if err != nil {
			return nil, err
		}
		a.T[symb] = m
	}

	return a, nil
}

// FancyString returns a decorated representation of an automaton
func (a Automaton) FancyString() string {
	var buf bytes.Buffer

	// Print the alphabet
	buf.WriteString("A = ")
	buf.WriteString(strings.Join(a.A, " "))
	buf.WriteString("\n")

	// Print the output vector
	fo := mat.Formatted(a.O, mat.Prefix(" "), mat.Squeeze())
	buf.WriteString(fmt.Sprintf("o = %.5g\n\n", fo))

	// Print the matrices
	for sym, m := range a.T {
		fo := mat.Formatted(m, mat.Prefix(" "), mat.Squeeze())
		buf.WriteString(fmt.Sprintf("T_%s = %.5g\n\n", sym, fo))
	}

	return buf.String()
}

// String representation of an automaton
func (a Automaton) String() string {
	var buf bytes.Buffer

	// Print the alphabet
	buf.WriteString(strings.Join(a.A, " "))
	buf.WriteString("\n")

	// Print the output vector
	fo := mat.Formatted(a.O, mat.Squeeze(), mat.FormatMATLAB())
	buf.WriteString(fmt.Sprintf("o = %.5g\n\n", fo))

	// Print the matrices
	for sym, m := range a.T {
		fo := mat.Formatted(m, mat.Prefix(" "), mat.Squeeze())
```

```
                buf.WriteString(fmt.Sprintf("T_%s = %.5g\n\n", sym, fo))
        }

        return buf.String()
}
```

Figure 15: automata/random.go

```
package automata

import (
        "math/rand"

        "github.com/0x0f0f0f/lwa-techniques/lin"
        "gonum.org/v1/gonum/mat"
)

// RandNatAutomaton generates a random automaton with weights on natural numbers
func RandNatAutomaton(syms, states, maxweight int) Automaton {
        // create the alphabet
        A := make([]string, syms)
        if syms <= 57 {
                for i := 0; i < syms; i++ {
                        A[i] = string(rune(i + 65))
                }
        }

        T := map[string]*mat.Dense{}
        for _, sym := range A {
                T[sym] = lin.RandIntDense(states, maxweight)
                lin.PokeHoles(T[sym], rand.Intn((states*states)/2))
        }

        O := lin.RandNatVec(states, maxweight)
        for lin.IsZero(O) {
                O = lin.RandNatVec(states, maxweight)
        }

        aut := Automaton{
                A: A,
                T: T,
                O: O,
                Dim: states,
        }

        return aut
}

// RandAutomaton generates a random automaton with weights on real numbers
func RandAutomaton(syms, states int, maxweight float64) Automaton {
        // create the alphabet
        A := make([]string, syms)
        if syms <= 57 {
                for i := 0; i < syms; i++ {
                        A[i] = string(rune(i + 65))
                }
        }

        T := map[string]*mat.Dense{}
```

```
	for _, sym := range A {
		T[sym] = lin.RandDense(states, maxweight)
		lin.PokeHoles(T[sym], rand.Intn((states*states)/2))
	}

	O := lin.RandVec(states, maxweight)
	for lin.IsZero(O) {
		O = lin.RandVec(states, maxweight)
	}

	aut := Automaton{
		A: A,
		T: T,
		O: O,
		Dim: states,
	}

	return aut
}
```

A.3 Pair, Relation and To-Do list Structures and Methods

Figure 16: automata/pair.go

```
// this file contains methods for handling vector pairs, exploiting the mat.Dense
// matrix type

package automata

import (
	"errors"

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

// NewPair creates a new pair of vectors
func NewPair(l, r *mat.VecDense) (*mat.Dense, error) {
	ld, _ := l.Dims()
	rd, _ := r.Dims()
	if ld != rd {
		return nil, errors.New("dimensions of vector do not match")
	}
	m := mat.NewDense(ld, 2, nil)

	for i := 0; i < ld; i++ {
		m.Set(i, 0, l.AtVec(i))
		m.Set(i, 1, r.AtVec(i))
	}
	return m, nil
}

// PairLeft returns left element of a vector pair
func PairLeft(p *mat.Dense) *mat.VecDense {
	return p.ColView(0).(*mat.VecDense)
}

// PairRight returns right element of a vector pair
func PairRight(p *mat.Dense) *mat.VecDense {
	return p.ColView(1).(*mat.VecDense)
```

```
}

// PairSub returns the subtraction of the elements of a vector pair
func PairSub(p *mat.Dense) *mat.VecDense {
	m, _ := p.Dims()
	sub := mat.NewVecDense(m, nil)
	sub.SubVec(PairLeft(p), PairRight(p))
	return sub
}

// PairCheck returns true if a matrix is a vector pair
func PairCheck(p *mat.Dense) bool {
	_, n := p.Dims()
	return n == 2
}

// PairEqs returns true if two pairs equal each other
func PairEqs(p, p1 *mat.Dense, tol float64) bool {
	m, _ := p.Dims()
	m1, _ := p1.Dims()
	// if the dimensions do not match, pairs are not equal
	if m != m1 || !PairCheck(p) || !PairCheck(p1) {
		return false
	}
	eq := true
	for i := 0; i < 2; i++ {
		eq = eq && mat.EqualApprox(p.ColView(i), p1.ColView(i), tol)
		eq = eq && mat.EqualApprox(p.ColView(i), p1.ColView(2-i), tol)
	}

	return eq
}
```

Figure 17: automata/pairstack.go

```
// stack data structure for real valued vector pairs, used in the HKC algorithm

package automata

import (
	"errors"

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

// NewPairStack creates a new pair stack, exploiting the mat.Dense matrix
// data type
func NewPairStack() *mat.Dense {
	m := mat.NewDense(1, 1, nil)
	m.Reset()
	return m
}

// PairStackSize returns the size of a pair stack
func PairStackSize(s *mat.Dense) int {
	if s.IsEmpty() {
		return 0
	}
	_, n := s.Dims()
```

```go
	return n
}

// PairStackPush pushes a pair into the stack
func PairStackPush(s *mat.Dense, p *mat.Dense) *mat.Dense {
	if s.IsEmpty() {
		return mat.DenseCopyOf(p)
	}
	s.Augment(s, p)
	return s
}

// PairStackPop pops a pair from the stack
func PairStackPop(s *mat.Dense) (*mat.Dense, error) {
	if s.IsEmpty() {
		return nil, errors.New("stack is empty")
	}
	m, n := s.Dims()
	if n%2 != 0 {
		return nil, errors.New("inconsisten stack: odd number of elements")
	}

	pair := s.Slice(0, m, n-2, n).(*mat.Dense)

	if n-2 > 0 {
		s = s.Slice(0, m, 0, n-2).(*mat.Dense)
	} else {
		s.Reset()
	}

	return pair, nil
}
```

Figure 18: automata/relation.go

```go
package automata

import (
	"errors"

	"github.com/0x0f0f0f/lwa-techniques/lin"
	"gonum.org/v1/gonum/mat"
)

// Relation represents a relation between sets of vectors of R^n.
// the relation is a congruence if it is an equivalence
// and is closed under linear combinations.
type Relation struct {
	s *mat.Dense // the set of pairs in the relations
	u *mat.Dense // generating set for the congruence closure
	size int // how many pairs
	dim int // number of rows
	tol float64
}

// NewRelation creates a new empty relation
func NewRelation(tol float64, dim int) Relation {
	s := mat.NewDense(dim, 1, nil)
	u := mat.NewDense(dim, 1, nil)
```

```
	s.Reset()
	u.Reset()
	return Relation{tol: tol, dim: dim, s: s, u: u}
}

// GetPair returns the pair in the relation at index i
func (r Relation) GetPair(i int) (*mat.Dense, error) {
	if i < 0 || i >= r.size {
		return nil, errors.New("index out of bounds")
	}
	return r.s.Slice(0, r.dim, i*2, (i*2)+2).(*mat.Dense), nil
}

// Has returns true if the relation contains the given pair of vectors.
// Computes in O(n). Could be done better by ordering.
func (r Relation) Has(p *mat.Dense) bool {
	m, _ := r.s.Dims()
	if m != r.dim && !PairCheck(p) {
		panic(errors.New("dimension mismatch"))
	}
	for i := 0; i < r.size; i++ {
		p1, err := r.GetPair(i)
		if err != nil {
			panic(err)
		}
		if PairEqs(p1, p, r.tol) {
			return true
		}
	}
	return false
}

// Add adds a pair of vectors to the relation
func (r *Relation) Add(p *mat.Dense) {
	// if the set already contains the pair (v, v'), return
	if r.Has(p) {
		return
	}
	r.dim++
	// add the pair (v,v') to the set
	if r.s.IsEmpty() {
		r.s = mat.DenseCopyOf(p)
	} else {
		r.s.Augment(r.s, p)
	}

	// add (v - v') result to the closure generating set
	sub := PairSub(p)
	subInU := false

	if r.u.IsEmpty() {
		r.u = mat.DenseCopyOf(sub)
		return
	}

	_, un := r.u.Dims()
	for j := 0; j < un; j++ {
		v := r.u.ColView(j).(*mat.VecDense)
		if mat.EqualApprox(v, sub, r.tol) {
```

```
                        subInU = true
                }
        }

        if !subInU {
                r.u.Augment(r.u, sub)
        }
}

// PairIsInCongruenceClosure checks
// if a pair of vectors is in a relation's congruence closure.
func (r Relation) PairIsInCongruenceClosure(p *mat.Dense) bool {
        // sub = v - v'
        sub := PairSub(p)

        // (v, v') c(R) iff v - v' U_R
        if r.u.IsEmpty() {
                return false
        }

        _, un := r.u.Dims()
        for j := 0; j < un; j++ {
                v := r.u.ColView(j).(*mat.VecDense)
                if mat.EqualApprox(v, sub, r.tol) {
                        return true
                }
        }

        return false
}

func (r Relation) String() string {
        return lin.StringMat(r.s) + lin.StringMat(r.u)
}
```

A.4 Algorithms

Figure 19: automata/hkc.go

```
// this file contains the implementation of the HKC procedure

package automata

import (
        "math"

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

// HKC checks the language equivalence of two state vectors for a
// given weighted automaton by building a congruence relation
func (a Automaton) HKC(v1, v2 *mat.VecDense) (bool, error) {
        rel := NewRelation(a.HKCTol, a.Dim)
        todo := NewPairStack()

        p, err := NewPair(v1, v2)
        if err != nil {
                return false, err
        }
```

```

	// insert (v1, v2) into the todo list
	todo = PairStackPush(todo, p)

	for !todo.IsEmpty() {
		// extract (v1', v2') from todo
		q, err := PairStackPop(todo)
		if err != nil {
			return false, err
		}

		if rel.PairIsInCongruenceClosure(q) {
			continue
		}

		o1 := a.GetOutput(PairLeft(q))
		o2 := a.GetOutput(PairRight(q))
		if math.Abs(o1-o2) > a.HKCTol {
			return false, nil
		}

		for _, sym := range a.A {

			w1 := a.ApplyTransition(sym, PairLeft(q))
			w2 := a.ApplyTransition(sym, PairRight(q))
			wp, err := NewPair(w1, w2)
			if err != nil {
				return false, err
			}

			PairStackPush(todo, wp)
		}

		// insert (v1', v2') into R
		rel.Add(q)
	}

	return true, nil
}
```

Figure 20: automata/backwards.go

```
// this file contains definitions for the backwards algorithm for
// computing the largest linear weighted bisimulation

package automata

import (
	"log"

	"github.com/0x0f0f0f/lwa-techniques/lin"
	"gonum.org/v1/gonum/mat"
)

// BackwardsPartitionRefinement computes and stores a basis for the
// complement of the largest linear weighted bisimulation of
// the linear weighted automaton. returns the condition number
func (a *Automaton) BackwardsPartitionRefinement() float64 {
	// i = 0
```

```
	lastBasis := mat.NewDense(a.Dim, 1, a.O.RawVector().Data)
	currBasis := lastBasis
	// condition number
	lastCond := 0.0

	for i := 1; i <= a.Dim; i++ {
		// \sum_{a \in A} T_a^T(R_i)
		for _, sym := range a.A {
			newBasis := a.ApplyTransposeTransitionBasis(sym, lastBasis)
			currBasis = lin.Union(currBasis, newBasis)
		}
		tmp, cond := lin.OrthonormalColumnSpaceBasis(currBasis, a.BPRTol)
		currBasis = tmp.(*mat.Dense)
		lastBasis = currBasis
		lastCond = cond
	}

	a.LLWBperp = currBasis
	// we could compute the orthogonal complement to find a basis of LLWB:
	// a.LLWB = lin.Complement(currBasis).(*mat.Dense)
	return lastCond
}

// BPREquivalence checks the equivalence of 2 vectors
// after a basis of the LLWB is computed through BPR,
func (a Automaton) BPREquivalence(v1, v2 *mat.VecDense) bool {
	if a.LLWBperp == nil {
		log.Fatalln("largest linear weighted bisimulation not computed for
		    automaton")
		return false
	}

	sub := mat.VecDenseCopyOf(v1)
	sub.SubVec(v1, v2)

	mul := mat.VecDenseCopyOf(sub)
	mul.Reset()
	mul.MulVec(a.LLWBperp.T(), sub)

	return lin.IsZeroTol(mul, a.BPRTol)
}
```

A.5 Random Batch Tests Package

Figure 21: randtest/data.go

```
// this file contains data structures relevant to tests

package randtest

import "fmt"

// Represent a sample result
const (
	TP int = iota // true positive
	TN // true negative
	FP // false positive
	FN // false negative
)
```

```

// BatchTestOptions represents options for running batch tests on automata
type BatchTestOptions struct {
	AutOptions *AutomatonTestOptions // initial automaton test options
	NumAutomata int // number of automata to generate and test
	Verbose bool
}

// Print batch test options
func (opt BatchTestOptions) Print() {
	if opt.Verbose {
		fmt.Println("========= BATCH OPTIONS ===========")
		fmt.Println(opt.NumAutomata, "automata")
		opt.AutOptions.Print()
	}
}

// BatchResult represents the result of many weighted
// language equivalence sample tests on many automata
type BatchResult struct {
	TP float64
	TN float64
	FP float64
	FN float64
	Null float64 // number of automata where ker(LLWB) is null
	Total float64 // Total number of tested automata
	Accuracy float64 // (TP+TN)/(TP+TN+FP+FN). percent of correctness
	Recall float64
	Precision float64
	F1 float64
	opt *BatchTestOptions
}

// Accumulate adds results of an automaton test to the results of a batch test
func (r *BatchResult) Accumulate(ar AutomatonResult) {
	r.TP += ar.TP
	r.TN += ar.TN
	r.FP += ar.FP
	r.FN += ar.FN
	if ar.Null {
		r.Null++
	}
	r.Total++
}

// ComputeStats computes relevant statistics on a batch test
func (r *BatchResult) ComputeStats() {
	T := r.TP + r.TN
	r.Accuracy = T / (T + r.FP + r.FN)
	r.Recall = r.TP / (r.TP + r.FN)
	r.Precision = r.TP / (r.TP + r.FP)
	r.F1 = 2 * ((r.Precision * r.Recall) / (r.Precision + r.Recall))
}

// Print results of a batch test
func (r BatchResult) Print() {
	if r.opt.Verbose {
		r.opt.Print()
		fmt.Println("========= RESULTS ===========")
```

```
			fmt.Println("LLWP is not empty for", float64(r.opt.NumAutomata)-r.Null,
				"automata")
			fmt.Printf("TP: %10d TN: %10d\n", int(r.TP), int(r.TN))
			fmt.Printf("FP: %10d FN: %10d\n", int(r.FP), int(r.FN))
			fmt.Printf("accuracy: %.20g\n", r.Accuracy)
			fmt.Printf("precision: %.20g\n", r.Precision)
			fmt.Printf("recall: %.20g\n", r.Recall)
			fmt.Printf("F1: %.20g\n", r.F1)
	}
}

// AutomatonTestOptions represents settings for generating random automata
type AutomatonTestOptions struct {
	NumStates int // Number of states in automata
	NumSymbols int // Number of symbols in alphabet
	NumSamples int // Number of samples
	MaxWeight int // Max modulo of the weight
	Mode string // Either "real" or "nat"
	BPRTol float64 // tolerance for BPR
	HKCTol float64 // tolerance for HKC
}

// Print automaton test options
func (opt AutomatonTestOptions) Print() {
	fmt.Println("weight kind:", opt.Mode)
	fmt.Println("max weight in modulo:", opt.MaxWeight)
	fmt.Println("number of samples per automata:", opt.NumSamples)
	fmt.Println("number of states:", opt.NumStates)
	fmt.Println("number of symbols:", opt.NumSymbols)
	fmt.Println("BPR tolerance:", opt.BPRTol)
	fmt.Println("HKC tolerance:", opt.HKCTol)
}

// AutomatonResult represents the result of many weighted
// language equivalence sample tests on a single automaton
type AutomatonResult struct {
	TP float64
	TN float64
	FP float64
	FN float64
	Null bool
}

// Accumulate adds results of a sample test to the results on an automaton test
func (r *AutomatonResult) Accumulate(kind int) {
	switch kind {
	case TP:
		r.TP++
	case TN:
		r.TN++
	case FP:
		r.FP++
	case FN:
		r.FN++
	}
}
```

Figure 22: `randtest/randtest.go`

```go
package randtest

import (
	"errors"
	"fmt"
	"math/rand"

	"github.com/0x0f0f0f/lwa-techniques/automata"
	"github.com/0x0f0f0f/lwa-techniques/lin"
	"gonum.org/v1/gonum/mat"
)

// BatchTest runs a batch test with given options
// and computes statistics at the end
func BatchTest(opt *BatchTestOptions) BatchResult {
	batchResults := BatchResult{opt: opt}

	for i := 0; i < opt.NumAutomata; i++ {
		fmt.Printf("testing automata %20d...\r", i)

		batchResults.Accumulate(TestRandAutomaton(opt.AutOptions))
	}

	fmt.Println()
	batchResults.ComputeStats()
	return batchResults

}

// TestRandAutomaton tests a single random automaton with the given options
func TestRandAutomaton(o *AutomatonTestOptions) AutomatonResult {
	var az automata.Automaton

	// choose between random real valued weights or natural
	switch o.Mode {
	case "real":
		az = automata.RandAutomaton(o.NumSymbols, o.NumStates,
		    float64(o.MaxWeight))

	case "nat":
		az = automata.RandNatAutomaton(o.NumSymbols, o.NumStates, o.MaxWeight)

	default:
		panic(errors.New("unknown mode"))
	}
	az.BPRTol = o.BPRTol
	az.HKCTol = o.HKCTol

	az.BackwardsPartitionRefinement()

	samples := make([]*mat.VecDense, o.NumSamples)
	randoms := make([]*mat.VecDense, o.NumSamples)

	// compute a basis of LLWB
	llwb := lin.Complement(az.LLWBperp, o.BPRTol).(*mat.Dense)
	_, dimLLWB := llwb.Dims()
	lin.CleanTolDense(llwb, o.BPRTol)

	if mat.Equal(llwb, mat.NewDense(o.NumStates, 1, nil)) {
```

```go
		return AutomatonResult{
			Null: true,
		}
	}

	autResult := AutomatonResult{Null: false}

	// generate language equivalent (in LLWB) and random pairs of vectors
	for i := range samples {
		samples[i] = lin.LinearCombination(llwb, lin.RandVec(dimLLWB, 100))
		randoms[i] = lin.RandVec(az.Dim, 100)
	}

	for i := range samples {
		j := rand.Intn(o.NumSamples)
		// test for vectors in span of LLWB
		for j == i {
			j = rand.Intn(o.NumSamples)
		}
		autResult.Accumulate(TestSamplePair(az, samples[i], samples[j]))
		// test for totally random vectors
		autResult.Accumulate(TestSamplePair(az, samples[i], samples[j]))

	}

	return autResult
}

// TestSamplePair tests two of vectors for language equivalence
// compares the results and returns the corresponding number for identifying
// if the result is true/false positive/negative
func TestSamplePair(az automata.Automaton, v1, v2 *mat.VecDense) int {
	BPReq := az.BPREquivalence(v1, v2)
	HKCeq, _ := az.HKC(v1, v2)

	if BPReq && HKCeq {
		return TP // true positive
	} else if !BPReq && !HKCeq {
		return TN // true negative
	} else if !BPReq && HKCeq {
		return FP // false positive
	} else {
		return FN // false negative
	}
}
```

Figure 23: `randtest/f1-tol.go`

```go
// contains test for F1 score related to tolerance values

package randtest

import (
	"fmt"
	"time"

	"github.com/alitto/pond"
	"github.com/jinzhu/copier"
	"gonum.org/v1/plot"
```

```
        "gonum.org/v1/plot/plotter"
        "gonum.org/v1/plot/plotutil"
        "gonum.org/v1/plot/vg"
)

// F1TolTask runs a test suite on random automata, collects F1 in relation
// to tolerance values and saves a plot in PDF and PNG format
func F1TolTask(fixedtol float64) {

        start := time.Now()

        aopts := &AutomatonTestOptions{
                NumStates: 4,
                NumSymbols: 2,
                NumSamples: 1000,
                MaxWeight: 2,
                Mode: "nat",
        }

        bopts := &BatchTestOptions{
                AutOptions: aopts,
                NumAutomata: 3000,
                Verbose: false,
        }

        tols := []float64{1e-22, 1e-21, 1e-20, 1e-19, 1e-18, 1e-17, 1e-16, 1e-15, 1e-14,
            1e-13, 1e-12, 1e-11, 1e-10, 1e-9, 1e-8, 1e-7, 1e-6}
        tolstr := []string{"1e-22", "1e-21", "1e-20", "1e-19", "1e-18", "1e-17", "1e-16",
            "1e-15", "1e-14", "1e-13", "1e-12", "1e-11", "1e-10", "1e-9", "1e-8", "1e-7",
            "1e-6"}

        // points on the graph
        ptsBoth := make(plotter.XYs, len(tols))
        ptsBPR := make(plotter.XYs, len(tols))
        ptsHKC := make(plotter.XYs, len(tols))

        //
            ==========================================================================================

        pool := pond.New(10, 100)

        // tests varying both tolerances
        pool.Submit(func() {
                fmt.Println("Running test varying both tolerances")
                for i, tol := range tols {
                        aopts.BPRTol = tol
                        aopts.HKCTol = tol

                        res := BatchTest(bopts)
                        res.Print()
                        ptsBoth[i].X = float64(i)
                        ptsBoth[i].Y = res.F1
                }
        })

        // tests varying HKC tolerance
        pool.Submit(func() {
                fmt.Println("Running test varying HKC tolerance")
                HKCAutomataOpts := &AutomatonTestOptions{}
```

```
			copier.Copy(HKCAutomataOpts, aopts)
			HKCAutomataOpts.BPRTol = fixedtol

			var HKCBatchOpts BatchTestOptions
			copier.Copy(&HKCBatchOpts, bopts)
			HKCBatchOpts.AutOptions = HKCAutomataOpts

			for i, tol := range tols {
				HKCAutomataOpts.HKCTol = tol

				res := BatchTest(&HKCBatchOpts)
				res.Print()
				ptsHKC[i].X = float64(i)
				ptsHKC[i].Y = res.F1
			}
		})

		// tests varying BPR tolerance
		pool.Submit(func() {
			fmt.Println("Running test varying BPR tolerance")

			BPRAutomataOpts := &AutomatonTestOptions{}
			copier.Copy(BPRAutomataOpts, aopts)
			BPRAutomataOpts.HKCTol = fixedtol

			var BPRBatchOpts BatchTestOptions
			copier.Copy(&BPRBatchOpts, bopts)
			BPRBatchOpts.AutOptions = BPRAutomataOpts

			for i, tol := range tols {
				BPRAutomataOpts.BPRTol = tol

				res := BatchTest(&BPRBatchOpts)
				res.Print()
				ptsBPR[i].X = float64(i)
				ptsBPR[i].Y = res.F1
			}
		})

		pool.StopAndWait()

		dur := time.Now().Sub(start)

		p, err := plot.New()
		if err != nil {
			panic(err)
		}

		p.Title.Text = fmt.Sprintf("Tests on %d automata, %d states, %d symbols, max |weight| = %d",
			bopts.NumAutomata,
			aopts.NumStates,
			aopts.NumSymbols,
			aopts.MaxWeight) +
			"\nTest took " + dur.String()
		p.X.Label.Text = "Tolerance"
		p.Y.Label.Text = "F1 Score"
		//p.X.Scale = plot.LogScale{}
		//p.X.Tick.Marker = plot.LogTicks{}
```

```
        p.NominalX(tolstr...)
        p.X.Tick.Width = vg.Points(0.5)
        p.X.Tick.Length = vg.Points(8)
        p.X.Width = vg.Points(0.5)

        plotutil.AddLinePoints(p,
                "Varying tolerance on both BPR and HKC", ptsBoth,
                "Varying tolerance on BPR, HKC tolerance set to "+fmt.Sprintf("%g",
                    fixedtol), ptsBPR,
                "Varying tolerance on HKC, BPR tolerance set to "+fmt.Sprintf("%g",
                    fixedtol), ptsHKC)

        // Save the plot to a PNG file.
        if err := p.Save(7*vg.Inch, 6*vg.Inch, fmt.Sprintf("paper/plots/f1-tol-%g.png",
            fixedtol)); err != nil {
                panic(err)
        }

        if err := p.Save(7*vg.Inch, 6*vg.Inch, fmt.Sprintf("paper/plots/f1-tol-%g.pdf",
            fixedtol)); err != nil {
                panic(err)
        }
}
```

Figure 24: main.go

```
package main

import (
        "math/rand"
        "time"

        "github.com/0x0f0f0f/lwa-techniques/randtest"
)

func check(err error) {
        if err != nil {
                panic(err)
        }
}

func main() {
        // defer profile.Start(profile.MemProfile).Stop()
        rand.Seed(time.Now().UnixNano())

        randtest.F1TolTask(1e-6)
        randtest.F1TolTask(1e-13)
        randtest.F1TolTask(1e-15 / 2)
        randtest.F1TolTask(1e-16)
        randtest.F1TolTask(1e-20)
}
```