Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Preservación de Normalidad en Transductores

(Preservation of Normality in Transducers)

Tesis de Licenciatura en Ciencias de la Computación

Elisa Orduna

LU.: 341/08

ordunaelisaorduna@gmail.com

Directores: Verónica Becher y Olivier Carton

Buenos Aires, 19 de Julio de 2018

PRESERVACIÓN DE NORMALIDAD EN TRANSDUCTORES

En esta tesis se plantea el problema de determinar si un transductor finito determinístico arbitrario devuelve una palabra normal siempre que recibe como entrada una palabra normal, es decir, si preserva normalidad. Una palabra infinita x es normal si todos los bloques de igual tamaño aparecen en x con la misma frecuencia asintótica. Trabajos anteriores caracterizan algunas familias de transductores que preservan normalidad: selectores de Agafonov, eliminación de todas las apariciones de un símbolo (en alfabetos con al menos tres símbolos), eliminación de finitos símbolos. Desarrollamos un algoritmo ideado por Olivier Carton que decide si un transductor determinístico preserva normalidad. Lo hace además en tiempo polinomial. En primer lugar, el algoritmo obtiene una descomposición del transductor en componentes fuertemente conexas y analiza todas aquellas que sean recurrentes por separado. Para cada componente se construye un autómata con pesos que permite calcular la frecuencia de una palabra finita arbitraria en la salida, suponiendo que la entrada es una palabra normal. Por último, se verifica que estas frecuencias sean las esperadas, utilizando una implementación polinomial del algoritmo de Schützenberger, propuesta por Crochemore. Además, presentamos un prototipo en Python que implementa el algoritmo.

Palabras clave: Aleatoriedad, Autómatas Finitos, Cadenas de Markov, Números Normales.

PRESERVATION OF NORMALITY IN TRANSDUCERS

In this thesis, we study the problem of deciding whether a given deterministic transducer outputs a normal word whenever it is fed with a normal word, in other words, whether it preserves normality or not. An infinite word x is normal if every block of the same length occurs in x with the same asymptotic frequency. Previous works characterize some families of transducers which preserve normality: Agafonov selectors, removal of all occurrences of a symbol (when alphabets contain at least three symbols), removal of a finite number of symbols. We develop an algorithm designed by Olivier Carton that decides if a given deterministic transducer preserves normality. Furthermore, this is done in polynomial time. First, the algorithm obtains a decomposition of the transducer into strongly connected components and analyzes those that are recurrent separately. For each component, it builds a weighted automaton, which allows to determine the frequency of an arbitrary finite word in the output, assuming that the input is a normal word. Finally, it checks that these frequencies match the expected ones, using a polynomial implementation of Schützenberger's algorithm, due to Crochemore. Additionally, we provide a prototype implementation in Python.

Keywords: Finite Automata, Markov Chains, Normal Numbers, Randomness.

AGRADECIMIENTOS

A Vero quiero agradecerle haberme recibido como tesista y cuidarme como a una hija. Es un momento de fragilidad y ansiedad, y tener toda esa atención y dedicación es un privilegio. ¡Gracias Vero!

A Olivier quiero agradecerle la suma paciencia y generosidad que tuvo hacia mí, nunca se quejó de explicarme dos veces las cosas y siempre me guió con mucho entusiasmo. Nunca tuve vergüenza de preguntar, y eso es porque siempre se encargaron de hacerme sentir cómoda.

A Santi Figueira y Sergio Abriola, el jurado, quiero agradecerles haber hecho el esfuerzo de evaluar mi tesis, con todo el trabajo que eso conlleva. Y también agradecerles haberlo hecho con cierto apremio, para que pudiera defender mi tesis con ambos directores presentes.

A veces escucho decir sobre Argentina '¡Qué país generoso!', y la verdad estoy de acuerdo. Tengo el privilegio de vivir en un país con educación pública y gratuita de excelencia, motor de oportunidades. Gracias por la oportunidad de formarme.

No quiero dejar pasar la oportunidad de agradecer a mis colegas docentes de las distintas materias en las que tuve oportunidad de participar, especialmente a mis amigos de TeLeng ¡que ya son mucho más que colegas! También a los alumnos, es enriquecedor de mil formas conocer a las nuevas camadas de estudiantes. Incluso estando un tiempo en la misma materia siempre hay una vuelta más de tuerca para darle, y en mí al menos esto sucede con las consultas de los chicos como disparador.

A mi familia por alentarme siempre, y especialmente a mis abuelas por hacerme sentir tan querida incondicionalmente y por estar orgullosas de mí cualquiera fuera el resultado de mis distintos proyectos.

¡A las poquitas amigas de hace mucho! A los amigos que ya no están, pero dejaron marca. A mis amigos matemáticos, mi incursión en matemática me dejó hermosas amistades. A mis amigos computadores.

A Pablo, por estar siempre, como compañero de cursada, como docente extraoficial, como amigo, como confidente, como compañero de vida.

A Tila.

CONTENTS

1. INTRODUCTION

Suppose that one were to flip a fair coin one million times. If the outcome turned out to be a sequence of one million heads, one would become suspicious of the coin fairness: one would expect about half of the tosses to be heads and about half to be tails. Now suppose that the outcome turned out to be a sequence HTHTHT...HT. Even though half of the tosses are heads, and half of the tosses are tails, one would still become suspicious that something strange is going on. How come there are no two consecutive heads? One would expect to obtain HH, HT, TH, TT with a frequency of about one fourth each. In general, one would expect that any two blocks of the same length have the same frequency. For example, HHHHH and THTTH should have the same frequency, namely they should appear about once every thirty-two times.

This observation motivates the formal definition of normality: an infinite word is *normal* if any two blocks of the same length have the same asymptotic frequency. More precisely, if the alphabet has b symbols, a finite word of length k should occur in a normal word with an asymptotic frequency of $\frac{1}{b^k}$. The notion of normality was introduced in 1909 by Borel [?] as an attempt to determine conditions for a number to be "random". In his seminal work, Borel showed that almost all words are normal. Nevertheless, there are not so many actual examples of words which have been proved to be normal.

The most famous example of a normal word is due to Champernowne [?], who showed in 1933 that the infinite word obtained from concatenating all the natural numbers (in their usual order):

$$0123456789101112131415161718192021222324252627282930\ldots$$

is normal in the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The same construction can be applied to other alphabets: concatenating all the finite words in increasing lexicographical order always yields a normal word.

Besicovitch [?] showed in 1935 that the word obtained from concatenating the *squares* of all the natural numbers, in increasing order:

$$149162536496481100121144169196225256289324361400441 4\ldots$$

is also normal, and later Copeland and Erdős [?] showed that the infinite word obtained from concatenating the *prime integers* in increasing order:

$$235711131719232931374143475359616771737983899710110 3\ldots$$

is normal as well.

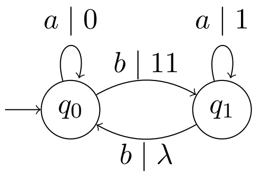
In 1992, Nakai and Shiokawa [?] proved that given a non-constant polynomial function $f(x)$ with real coefficients such that $f(x) > 0$ for all $x > 0$, the infinite word:

$$\lfloor f(1) \rfloor \lfloor f(2) \rfloor \lfloor f(3) \rfloor \ldots$$

that results from concatenating the integer part of $f(i)$ for $i = 1, 2, \ldots$ is normal in the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Nakai and Shiokawa's result simultaneously generalizes Champernowne's, by taking $f(x) = x$, and Besicovitch's, by taking $f(x) = x^2$.

Normal numbers are intimately related with *finite automata*. In 1971, Schnorr and Stimm proved that a word is normal if and only if no martingale described by a finite-state automaton succeeds in making unbounded profit [?]. Another characterization of normality in terms of finite automata establishes that an infinite word is normal if and only if it is *incompressible* by a finite-state transducer [?, ?, ?].

A finite-state transducer is a finite-state automaton in which each transition, besides consuming an input symbol, also produces a word as part of the output. The following is an example of a two-state transducer:

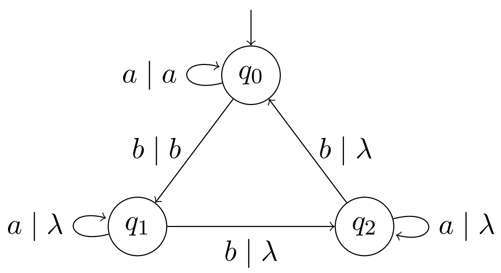


In this example, each transition is labeled with an input symbol from the alphabet $\{a, b\}$ and an output word in the alphabet $\{0, 1\}$. If this transducer is fed with *ababa* as an input, starting from the initial state q_0, the output is 01110.

Problems that can be posed as a yes or no question are called *decision problems*, for example: Given a transducer, does it preserve normality? Given a program and an input, does the program ever finish running? We say that a decision problem is decidable if there exists an algorithm that solves the question in the general case. Additionally, we say that a decision problem is undecidable if it cannot be decided in general with an algorithm. Many questions that are decidable for finite-state automata turn out to be undecidable for finite-state transducers. For instance, the problem of determining whether two given transducers are equivalent, that is, whether they describe the same input/output relation, is undecidable. In fact, it is not even possible to decide, given two transducers, whether there exists an input word for which they yield the same output [?]. We are interested in studying the behaviour of a finite-state transducer when the input is a normal word.

The main problem that motivates this thesis is to determine whether a given finite-state transducer yields a normal word as an output, when it is fed a normal word as an input. When this happens, we say that the transducer in question *preserves normality*. Is this problem decidable? Recall that an infinite word x is normal if every finite word of any length k in an alphabet with b symbols, occurs in x with an asymptotic frequency of $\frac{1}{b^k}$. Answering the question means analyzing the infinite outputs of all infinite normal words and the frequencies of all (infinitely many) finite words in them. Since the question is essentially infinitary, it is not obvious that the problem is indeed decidable.

It is known that *Agafonov selectors* preserve normality [?]. An Agafonov selector is a deterministic transducer in which each state can be of one of two types: type I and type II. Transitions going out from a state of type I behave as the identity function, that is, they produce as output the same symbol that they consume. Transitions going out from a state of type II only output λ, that is, they delete the symbol consumed. For instance, in the following Agafonov transducer, the state q_0 is of type I, and the states q_1, q_2 are of type II:



Checking whether a given transducer is an Agafonov selector is easy, in fact it can be done in linear time. It has been recently proved that deleting all the occurences of a fixed symbol from a normal word also yields a normal word [**?**]. For example, removing the symbol 1 from Champernowne's constant results in the following infinite word:

$$023456789023456789202222324252627282930\ldots$$

which turns out to be normal in the alphabet $\{0, 2, 3, 4, 5, 6, 7, 8, 9\}$. It is clear that changing a finite number of symbols also preserves normality. Observe that normality is a property about the *asymptotic* behaviour of an infinite word, so making any change in a finite prefix does not change its status of being normal.

In this thesis we study an algorithm that given a deterministic transducer, determines whether it preserves normality in polynomial time. We also give a prototype implementation in Python. In the following section, we present the formal notions of *normality* and *transducer*, which are crucial to be able to state the goal more precisely.

1.1 The problem

In this section we provide the formal definitions required to determine whether an input-deterministic transducer preserves normality. We also present the definition of weighted automata, on which the algorithm relies as an essential tool.

1.1.1 Normality

Before giving the formal definition of normality, let us introduce some simple definitions and notations. Let A be a finite set of symbols that we refer to as the alphabet. We write A^ω for the set of all infinite words in the alphabet A, and A^* for the set of all finite words. The size of A is written $|A|$ and the length of a finite word w is denoted by $|w|$. The positions of finite and infinite words are numbered starting at 1. To denote the symbol at position i of a word w we write $w[i]$, and to denote the substring of w from position i to j inclusive we write $w[i \ldots j]$. The empty word is denoted by λ. Given two words w and v in A^* , the number $|w|_v$ of occurrences of v in w is defined by:

$$|w|_v = |\{i : w[i \ldots i + |v| - 1] = v\}|$$

For example, $|aaaaa|_{aa} = 4$.

Definition 1 (Frequency of a word)**.** Given a finite word $w \in A^*$ and an infinite word $x \in A^\omega$, we define the *frequency of w in x* as

$$\mathit{freq}(x, w) = \lim_{n \to \infty} \frac{|x[1 \dots n + |w|]|_w}{n}$$

where $w = b_1 \dots b_n$.

Definition 2 (Normal word)**.** An infinite word $x \in A^\omega$ is *normal* in the alphabet A if for every word $w \in A^*$:

$$\mathit{freq}(x, w) = \frac{1}{|A|^{|w|}}$$

An alternative definition of normality can be given by counting *aligned* occurrences, and it is well-known that they are equivalent (see for example [**?**]).

1.1.2 Deterministic Transducers

Our main concern is to analyze Deterministic Transducers, which constitute the input of the algorithm. In this section we give the formal definition of a transducer, some relevant types of transducers and a *run* in a transducer. To determine whether a transducer preserves normality means to determine whether the output of every run whose input is a normal word is also normal.

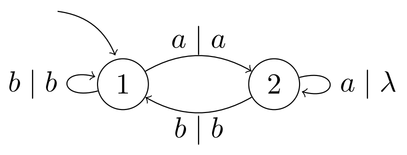
Definition 3 (Transducer)**.** A *transducer* $\mathcal{T}$ is a tuple $\langle Q, A, B, \delta, I \rangle$, where

- Q is a finite set of states,
- A and B are the input and output alphabets respectively,
- $\delta \subseteq Q \times A \times B^* \times Q$ is a finite transition relation,
- $I \subseteq Q$ is the set of initial states.

For example, the following is a transducer that compresses blocks of consecutive as into a single a:

$$\mathcal{T} = \langle \{1, 2\}, \{a, b\}, \{a, b\}, \{(1, b, b, 1), (1, a, a, 2), (2, a, \lambda, 2), (2, b, b, 1)\}, \{1\} \rangle$$

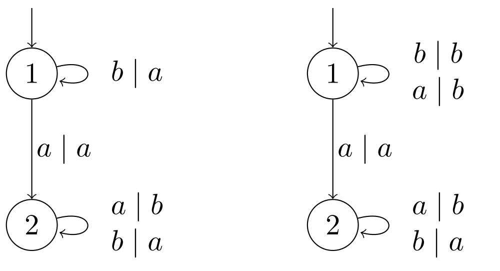
As usual, a transducer can be represented graphically. Each state is represented with a node, a transition $(q, v, w, q') \in \delta$ is represented with a labeled arrow $q \xrightarrow{v|w} q'$. Each initial state is marked with an unlabeled arrow pointing to it, with no origin. The transducer $\mathcal{T}$ above can be depicted as follows:



From now on, we represent transducers graphically when appropiate. We usually assume that there is a single initial state, whose number is 1, and there is no need to mark it with an unlabeled arrow.

Definition 4 (Deterministic transducer)**.** A transducer $\mathcal{T}$ is called an *input-deterministic transducer*, or *deterministic transducer* for short, if:

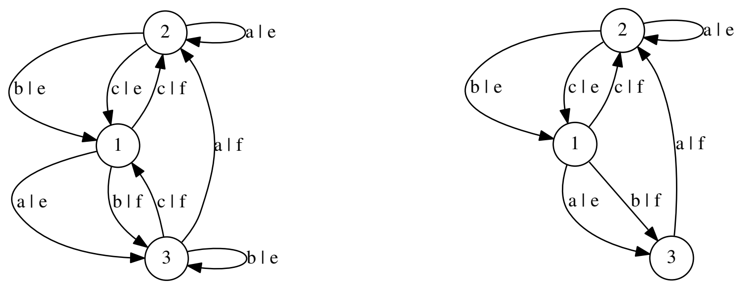
- It has a single initial state, that is, $|I| = 1$.
- If there exists transitions $p \xrightarrow{a|v} q$ and $p \xrightarrow{a|v'} q'$, then $q = q'$ and $v = v'$.



(a) Deterministic *(b)* Non deterministic

Fig. 1.1: A deterministic transducer and a non deterministic transducer. Second one is non deterministic since there are two different transitions leaving state 1 and consuming a.

Definition 5 (Complete transducer)**.** A transducer $\mathcal{T}$ is said to be *complete* if for each symbol $a \in A$ and each state $p \in Q$ there is a transition from p and consuming a, in other words there exist a word $w \in B^*$ and a state $q \in Q$ such that $p \xrightarrow{a|\delta} q$.



(a) Complete *(b)* Incomplete

Fig. 1.2: A complete transducer and an incomplete transducer. Second one is incomplete since there are no transitions leaving state 3 and consuming b.

Definition 6 (Deterministic complete transducer)**.** A transducer $\mathcal{T}$ is said to be an *input-deterministic complete transducer* or *deterministic complete transducer* for short, if it is both *input-deterministic* and *complete.*

Definition 7 (Run)**.** A finite (respectively infinite) *run* in $\mathcal{T}$ is a finite (respectively infinite) sequence of consecutive transitions,

$$q_0 \xrightarrow{a_1|v_1} q_1 \xrightarrow{a_2|v_2} \dots q_{n-1} \xrightarrow{a_n|v_n} q_n$$

We may refer to $a_1 a_2 \dots a_n$ as the *input* or *label of the run.*

For example, $1 \xrightarrow{a|e} 3 \xrightarrow{a|f} 2 \xrightarrow{a|e} 2 \xrightarrow{b|e} 1 \xrightarrow{c|f} 2$ is a run in Figure 1.2a, with input $aaabc$ and output $efeef$.

When $\mathcal{T}$ is a deterministic complete transducer, we refer to the output word of the only run in $\mathcal{T}$ starting in the initial state and consuming an infinite word x as $\mathcal{T}(x)$. We may refer to $\mathcal{T}(x)$ as just the output when consuming x. Note that though x is infinite, $\mathcal{T}(x)$ may be finite.

Definition 8 (Preserves normality)**.** We say that a deterministic transducer $\mathcal{T}$ *preserves normality* if and only if for all normal words x, $\mathcal{T}(x)$ is also normal.

1.1.3 Weighted Automata

Though *weighted automata* are not part of the problem itself, they are a key part of the solution. The question of whether a deterministic transducer preserves normality can be answered by conveniently building a weighted automaton that allows to calculate $freq(\mathcal{T}(x), w)$ for every $w \in B^*$.

Definition 9 (Weighted automaton)**.** A *weighted automaton* $\mathcal{A}$ is a tuple $\langle Q, B, \Delta, I, F\rangle$, where

- $I : Q \to \mathbb{R}$ is a function that assigns each state an initial weight
- $F : Q \to \mathbb{R}$ is a function that assigns each state an final weight
- $\Delta : Q \times B \times Q \to \mathbb{R}$ is a function that assigns each transition a weight

Figure 1.3 displays an example of a weighted automaton. A weighted automaton is represented graphically with the following convention. States are labeled with ${}_{[w_i]}q_{[w_f]}$ where q is the name of the state, w_i is the initial weight and w_f is the final weight. Transitions are labeled with $x_{[w]}$, where x is an input symbol, and w is the weight of the transition.

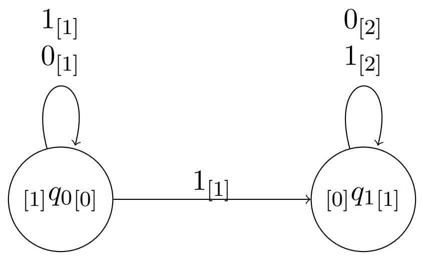


Fig. 1.3: An example of a weighted automaton

Definition 10 (Run)**.** A *run* in $\mathcal{A}$ is a finite sequence of consecutive transitions, $q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$. We refer to $b_1 \dots b_n$ as the *label of the run.*

Definition 11 (Weight of a run)**.** The *weight of a run* $q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n$ in $\mathcal{A}$ is the product of the weights of its n transitions times initial and final weights:

$$weight_{\mathcal{A}}(q_0 \xrightarrow{b_1} q_1 \xrightarrow{b_2} \dots \xrightarrow{b_n} q_n) = I(q_0) \times \Delta(q_0, b_1, q_1) \times \dots \Delta(q_{n-1}, b_n, q_n) \times F(q_n)$$

The weight of the empty run is 1.

Definition 12 (Weight of a word)**.** Given a weighted automaton $\mathcal{A} = \langle Q, B, \Delta, I, F \rangle$ and a word $w \in B^*, w = b_1 \dots b_n$, we define the *weight of the word* w as the sum of weights of all runs labeled with w:

$$weight_{\mathcal{A}}(w) = \sum_{\gamma \in R(w)} weight_{\mathcal{A}}(\gamma)$$

where $w = b_1 \dots b_n$ and $R(w) = \{\gamma \mid \gamma = q_0 \xrightarrow{b_1} q_1 \dots \xrightarrow{b_n} q_n \text{ is a run in } \mathcal{A}\}$

The run $q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_1$ in the weighted automaton from Figure 1.3 is labeled with 1010 and its weight is $1 \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot 1 = 8$. The run $q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_1$ in the weighted automaton from Figure 1.3 is labeled with 1010 and its weight is $1 \cdot 1 \cdot 1 \cdot 1 \cdot 2 \cdot 1 = 2$. Note that every run starting on q_1 has weight zero, as the initial weight of q_1 is 0. Moreover, any other run labeled with 1010 starting on q_0, besides the two runs we have already discussed, must have weight 0, as it necessarily finishes on q_0, which has final weight 0. Hence, $weight_{\mathcal{A}}(1010) = 8 + 2 = 10$. Note that the weight of a word in this automaton computes its value if interpreted as a number in base 2.

1.2 Structure of this thesis

The remainder of this thesis is organized as follows:

- In Chapter 2, we describe an algorithm for deciding whether a deterministic transducer preserves normality. We mention the facts that make the algorithm correct, and we study its theoretical complexity.
- In Chapter 3, we describe the details of our implementation of the algorithm in Python, including the test cases.
- In Chapter 4, we conclude.

2. NORMALITY-PRESERVING TRANSDUCERS

In this chapter, we describe a solution for the problem of deciding whether a deterministic transducer preserves normality. We focus in deterministic complete transducers in which all states are *reachable* from the initial state: states that are not reachable from the initial state have no impact in any translation, and transducers that are not complete do not preserve normality. From now on, we assume transducers to be deterministic and complete, and we assume each of its states to be reachable from the initial state.

The algorithm has two phases. Firstly, the algorithm finds the *recurrent strongly connected components* of the transducer. A transducer may have one or more components, each of which is a smaller transducer. Secondly, the algorithm relies on the observation that the original transducer preserves normality if and only if all of its components do. As a consequence, it suffices to analyze each component independently, to check whether it preserves normality or not. We argue that a deterministic complete transducer can have at most a linear number of recurrent strongly connected components, and that the cost of analyzing each component is polynomial in time, thus obtaining a polynomial algorithm.

This chapter is organized as follows:

- In section 2.1, we describe the algorithm itself.
- In section 2.2, we mention the facts that prove the algorithm to be correct.
- In section 2.3, we study its worst-case asymptotic time complexity.

2.1 The Algorithm

In this section we describe the main algorithm, which has two phases, as mentioned before:

Input: $\mathcal{T} = \langle Q, A, B, \delta, I \rangle$ an input-deterministic complete transducer.
Output: `True` if $\mathcal{T}$ preserves normality, `False` if $\mathcal{T}$ does not preserve normality.
Procedure:

1. Find the set $\{S_1, \ldots, S_k\}$ of recurrent strongly connected components of $\mathcal{T}$.
2. Analyze if each $S_i \in \{S_1, \ldots, S_k\}$ preserves normality.
   If they all do, return `True`, otherwise return `False`.

2.1.1 Decomposition into recurrent strongly connected components

We now present the definitions of strongly connected components and recurrent strongly connected components, together with some examples.

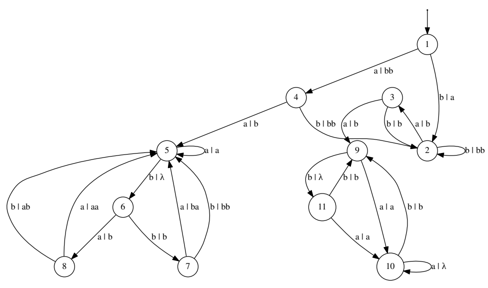
Definition 13 (Strongly connected components). Let $\mathcal{T} = \langle Q, A, B, \delta, I \rangle$ be a deterministic complete transducer. If there exists a run

$$p = r_0 \xrightarrow{a_1|v_1} r_1 \xrightarrow{a_2|v_2} r_2 \ldots \xrightarrow{a_n|v_n} r_n = q$$

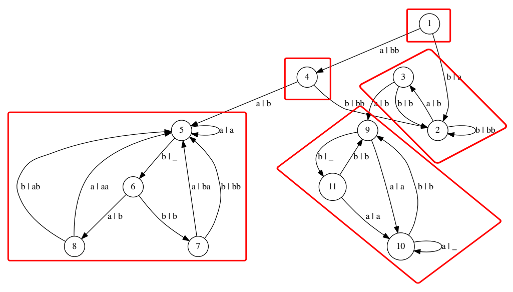
for some $n \geq 0$, we say that q is *reachable* from p. A subset $S \subseteq Q$ of the set of all states is called a *strongly connected component* (SCC) if S is a maximal set verifying the following property: for any two states $p, q \in S$, the state q is reachable from p.

Definition 14 (Recurrent SCC)**.** A strongly connected component S is called *recurrent* if no transition leaves it.

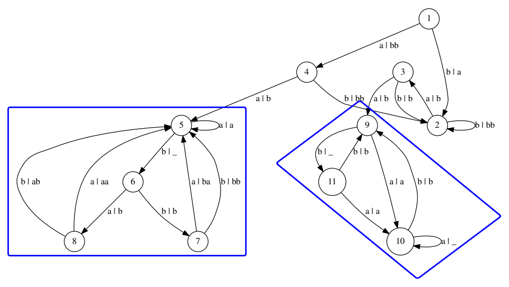
For example, in Fig. 2.1a, we exhibit a deterministic complete transducer $\mathcal{T}$, in Fig. 2.1b we mark its strongly connected components, and in Fig. 2.1c we mark its recurrent strongly connected components.



(a) A deterministic complete transducer $\mathcal{T}$.



(b) Strongly connected components of $\mathcal{T}$.



(c) Recurrent strongly connected components of $\mathcal{T}$.

Fig. 2.1: An example of a transducer, its SCCs and its recurrent SCCs.

In order to find the strongly connected components of the transducer $\mathcal{T}$, we apply Kosaraju's algorithm [**?**, Section 22.5], which finds all the SCCs of an arbitrary directed graph. Only the recurrent SCCs are kept for the next phase of the analysis. Observe that since $\mathcal{T}$ is complete, its recurrent strongly connected components are also complete.

2.1.2 Preservation of normality

We now present how to analyze preservation of normality in a recurrent strongly connected component. In later sections we explain each step in more detail.

For each recurrent strongly connected component S of $\mathcal{T}$, we analyze the transducer $\mathcal{T}$'$= \langle S, A, B, \delta', \{q_i\}\rangle$ induced by S. It is clear that $\mathcal{T}'$ has the same input and output alphabets as $\mathcal{T}$, the set of states is precisely $S \subseteq Q$, and the transition function δ' is δ restricted to S. Now it may not be clear what the initial state q_i should be. For the purpose of analyzing the output of a normal word, and since we are working on a strongly connected component, any choice of initial state can be made. Let us write f to refer to the function $f : X \subseteq A^\omega \to B^\omega$ realized by $\mathcal{T}$'. The algorithm consists on three main steps:

1. We apply a transformation called *normalization* to the transducer $\mathcal{T}'$. As a result, we obtain a normalized transducer $\mathcal{T}''$ which still realizes f.
2. We build a weighted automaton that realizes the function g that calculates the frequency of a finite word in the output thrown by $\mathcal{T}''$ when fed with a normal word.
3. We use the weighted automaton to check whether the frequency of every finite word in B^* is the expected one for the output to be normal, i.e. we check that the following holds for every word $w \in B^*$:
$$g(w) = \frac{1}{|A|^{|w|}}$$

In the rest of this chapter, we refer to $\mathcal{T}$' as just $\mathcal{T}$.

2.1.2.1 Transducer normalization

Normalizing consists in transforming the transducer to avoid having transitions of the form $p \xrightarrow{a|w} q$ where $|w| \geq 2$. This requires introducing λ-*transducers*, which are transducers extended to allow λ-*transitions*, and the notion of *normal form.*

Definition 15 (λ-transducer)**.** A λ-*transducer* $\mathcal{T}$ is a tuple $\langle Q, A, B, \delta, I\rangle$, where:

- Q is a finite set of states,
- A and B are the input and output alphabets respectively,
- $\delta \subseteq Q \times (A \cup \{\lambda\}) \times B^* \times Q$ is a finite transition relation,
- $I \subseteq Q$ is the set of initial states.

We refer to transitions $p \xrightarrow{\lambda|v} q$ as λ-*transitions.*

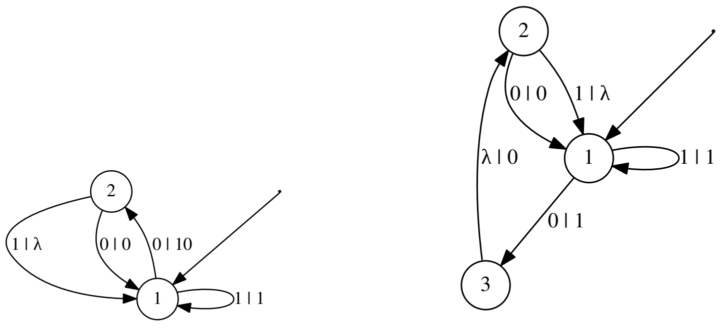
Definition 16 (Normal form)**.** A λ-transducer $\mathcal{T} = \langle Q, A, B, \delta, I\rangle$ is in *normal form* if the three following conditions hold:

1. for every transition $p \xrightarrow{a|w} q$ we have that $|w| \leq 1$,
2. for every transition $p \xrightarrow{\lambda|w} q$ we have that $|w| = 1$,
3. for every state $q \in Q$, either:
    - there is either a single λ-transition leaving q, and no other transitions,
    - or there are exactly $|A|$ transitions leaving q, each labeled with a distinct symbol from A.

To obtain an equivalent λ-transducer in normal form each transition $p \xrightarrow{a|w} q$ in $\mathcal{T}$ such that $|w| \geq 2$ is *replaced* by n transitions:

$$p \xrightarrow{a|b_1} q_1 \xrightarrow{\lambda|b_2} q_2 \dots \xrightarrow{\lambda|b_{n-1}} q_{n-1} \xrightarrow{\lambda|b_n} q$$

where $w = b_1 \dots b_n$ and $q_1, q_2, \dots, q_{n-1}$ are new states.



(a) Original transducer. *(b)* Normalized transducer.

Fig. 2.2: In this example, normalization adds a new state.

Note that the resulting normalized transducer may not be deterministic for finite inputs, in the sense that there may be more than one possible output for a single input, but it is still deterministic for infinite inputs.

2.1.2.2 Construction of the weighted automaton

At this point, the transducer $\mathcal{T} = \langle Q, A, B, \delta, I \rangle$ is normalized so the output in each transition is never longer than a symbol. The next goal is to produce a *weighted automaton* $\mathcal{A} = \langle Q, B, \delta', I', F' \rangle$, to allow us to know, for any finite word, its frequency in the output of a normal word. The weighted automaton is built to have the same states as $\mathcal{T}$, and transitions relating an output symbol with the frequency it has in the translation of any normal word. The transitions of $\mathcal{A}$ and their corresponding weights are obtained by assigning weights to the transitions of $\mathcal{T}$. Once each transition has been assigned a weight, we interpret $\mathcal{A}$ as a Markov chain, and we use its stochastic matrix and its associated stationary distribution to define the initial weights for every state in $\mathcal{A}$. The final weight is defined to be 1 for every state in $\mathcal{A}$. Built in this way, the weighted automaton allows us to calculate the frequency of a finite word in the output produced by $\mathcal{T}$ when fed a normal word.

Defining the transitions of the automaton and their weights. Recall that the states of the transducer $\mathcal{T}$ have either all possible transitions with symbols from A or a single λ-transition, since $\mathcal{T}$ has been normalized. If there are no transitions with output λ then all transitions output a single output symbol. In that case, it is fairly clear how often a symbol in B is produced as the output from a given state, when consuming a normal word. To calculate this frequency, we can simply look at the transitions that have that symbol as output and add their weights. The presence of λ as output makes this not so obvious, since many transitions with empty output could be taken before producing any output symbol. To deal with this issue, we consider separately all possible runs that start in a fixed state and output λ, and their corresponding weights. Later we show that this can be effectively computed by solving a system of linear equations.

We assign weights to the transitions in $\mathcal{T}$ as follows: we assign weight $1/|A|$ to transitions that are labeled with a symbol from A, and 1 to λ-transitions, representing the frequency with transitions from each state are taken when consuming a normal word. We define the weight of a run in $\mathcal{T}$. This allows us to assign weight to transitions of the form $p \xrightarrow{b} q$ in $\mathcal{A}$, by combining the weights of every possible run r from p that output λ with the weight of the single transition that connects the last state of the run r with q and outputs b.

Given a word $w = a_1 a_2 \dots a_n$ and a run $q_0 \xrightarrow{a_1|v_1} q_1 \xrightarrow{a_2|v_2} \dots \xrightarrow{a_n|v_n} q_n$ in $\mathcal{T}$, the *weight of the run* in $\mathcal{T}$ is the product of the weights of its n transitions:

$$weight_{\mathcal{T}}(q_0 \xrightarrow{a_1|v_1} q_1 \xrightarrow{a_2|v_2} \dots \xrightarrow{a_n|v_n} q_n) = \prod_{i=1}^{n} weight_{\mathcal{T}}(q_{i-1} \xrightarrow{a_i|v_i} q_i)$$

The weight of the empty run is 1.

Given two states $p, q \in Q$, we write $X_{p,q}$ to denote the set of runs in $\mathcal{T}$ from p to q with empty output:

$$X_{p,q} = \{q_0 \xrightarrow{a_1|\lambda} q_1 \xrightarrow{a_2|\lambda} \dots \xrightarrow{a_1|\lambda} q_n : q_0 = p, q_n = q\}$$

We define $x_{p,q}$ as the sum of weights of runs in $\mathcal{T}$ from p to q with empty output:

$$x_{p,q} = \sum_{\gamma \in X_{p,q}} weight_{\mathcal{T}}(\gamma)$$

Observe that $x_{p,q}$ either converges to a non-negative real number or tends to infinity $(+\infty)$, since all the terms of the sum are non-negative.

Computation of $x_{p,q}$. To effectively calculate each $x_{p,q}$, we show that the associated matrix is the solution to a linear system of equations. For that purpose, consider the matrix $M \in \mathbb{R}^{Q \times Q}$ whose entries are given by $x_{p,q}$ for each pair of states (p, q). Recall that $x_{p,q}$ is the sum of the weights of runs in $\mathcal{T}$ that go from p to q and have empty output, of *arbitrary* length. Moreover, let $E \in \mathbb{R}^{Q \times Q}$ be the matrix whose entry at position (p, q) is the sum of the weights of runs from p to q with empty output, of length *exactly* 1, that is:

$$E_{p,q} = \sum_{p \xrightarrow{a|\lambda} q} weight_{\mathcal{T}}(p \xrightarrow{a|\lambda} q)$$

The matrix M can be obtained from:

$$M = Id + E + E^2 + E^3 + \dots = \sum_{k \geq 0} E^k$$

where for each $k \geq 0$, the entry of E^k at position (p, q) is the sum of the weights of runs from p to q with empty output whose length is *exactly* k. Now let $E^* = \sum_{k \geq 0} E^k$, and note that the following equalities hold:

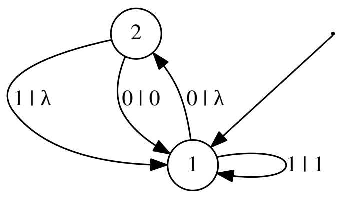
$$\begin{aligned} E^* &= Id + (E + E^2 + E^3 + \dots) \\ &= Id + (Id + E + E^2 + E^3 + \dots) \cdot E \\ &= Id + E^* \cdot E \end{aligned}$$

Hence, E^* is a solution to the equation $X = Id + X \cdot E$, which can be rewritten as follows:

$$X \cdot (Id - E) = Id$$

Since E^* is an infinite sum of matrices with non-negative coefficients, each entry of E^* can either converge to a non-negative number or tend to infinity. If it is the case that there is at least one infinite entry in E^*, we can already answer that $\mathcal{T}$ does not preserve normality. In fact, this means that there exists a normal word for which the output is finite.

For example, the matrices E and E^* are the following for the transducer below:



$$\begin{array}{cc} E & E^* \\ \begin{bmatrix} 0 & 1/2 \\ 1/2 & 0 \end{bmatrix} & \begin{bmatrix} 4/3 & 2/3 \\ 2/3 & 4/3 \end{bmatrix} \end{array}$$

From E^* and $\mathcal{T}$ we build the transitions of $\mathcal{A}$ over the alphabet B, with weights:

$$weight_{\mathcal{A}}(p \xrightarrow{b} q) = \sum_{r \in Q, a \in A} x_{p,r} \cdot weight_{\mathcal{T}}(r \xrightarrow{a|b} q)$$

For each $b \in B$, we define $N_b \in \mathbb{Q}^{Q \times Q}$ as the matrix whose entry at position (p, q) is the sum of the weights of all transitions from p to q whose output is b, that is:

$$(N_b)_{p,q} = \sum_{a \in A \cup \{\lambda\}} weight_{\mathcal{T}}(p \xrightarrow{a|b} q)$$

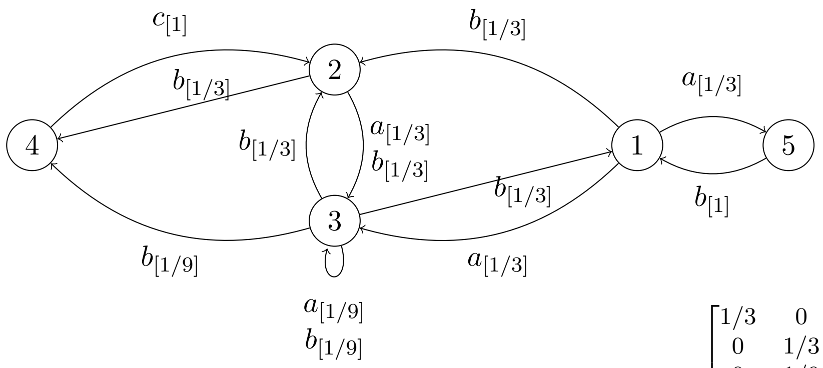
Having calculated E^* and N_b for each $b \in B$, we can effectively calculate $weight_{\mathcal{A}}(p \xrightarrow{b} q)$ as it is the (p, q)-entry of the product $E^* \cdot N_b$.

It only remains to define the initial and final weights for every state of the weighted automaton $\mathcal{A}$. So far, we have an automaton with weights in transitions. Observe that the sum of the weights of the transitions leaving any state q is always 1. Let us consider $\mathcal{A}$ as a Markov chain[1] to define its initial weights. We can do this since the weights on transitions are positive, and the sum of weights of transitions leaving a state is 1. The corresponding matrix $T \in [0, 1]^{Q \times Q}$ is given by the entries:

$$T_{p,q} = \sum_{b \in B} weight(p \xrightarrow{b} q)$$

[1] To the ends of this thesis, it suffices to say that a Markov chain is a connected graph whose transitions have probabilities associated, hence the probabilities of transitions leaving a state add up to 1.

Example 17. An automaton with weights and its associated matrix T:



$$\begin{array}{c} T \\ \begin{bmatrix} 1/3 & 0 & 1/3 & 1/3 & 0 \\ 0 & 1/3 & 2/3 & 0 & 0 \\ 0 & 1/9 & 2/9 & 1/3 & 1/3 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

As defined, the matrix T happens to be a stochastic matrix:

Definition 18. A *stochastic matrix* (also known as Markov matrix) is a square matrix $P \in (\mathbb{R}_{\geq 0})^{n \times n}$ such that each row adds up to 1.

Moreover, to define the initial weights of the automaton, we need the notion of stationary distribution of a stochastic matrix:

Definition 19. Let P be a stochastic matrix. A vector $\pi = (x_1, \ldots, x_n)$ is called a *stationary distribution of* P if $\pi \cdot P = \pi$ and moreover $\sum_{i=1}^{n} x_i = 1$.

The initial weights of the states of the automaton $\mathcal{A}$ are given precisely by the (unique) stationary distribution of T. The final weights of the states of the automaton $\mathcal{A}$ are all set to 1. Built in this way, $\mathcal{A}$ can be used to effectively calculate $\mathit{freq}(\mathcal{T}(x), w)$ for any normal word $x \in A^\omega$, and for any finite word $w \in B^*$ as follows. Consider π and for every $b \in B$, the matrix $T(b) \in \mathbb{R}_{\geq 0}^{n \times n}$ containing the weights of transitions with label b: $(T(b))_{p,q} = \mathit{weight}_{\mathcal{A}}(p \xrightarrow{b} q)$. For each word $w = b_1 \ldots b_n$, the vector obtained from multiplying $\pi \cdot T(b_1) \cdot T(b_2) \ldots T(b_n)$ contains for each state the frequency with which the word w is the next output, when the run from consuming a normal word goes through that state. The sum of that vector is thus the frequency of w in the output of a normal word, in other words, the product of $\pi \cdot T(b_1) \cdot T(b_2) \ldots T(b_n) \cdot (1 \ldots 1)^t$ is the frequency of w in the output of a normal word.

2.1.2.3 Comparison against the expected frequencies

At this point we have an automaton $\mathcal{A}$ that calculates effectively $\mathit{freq}(\mathcal{T}(x), w)$, for any normal word $x \in A^\omega$, and for any finite word $w \in B^*$. Now everything is set to compare, for every word $w \in B^*$, this frequency to the expected one for $\mathcal{T}(x)$ to be normal. We do this comparison by first building a weighted automaton $\mathcal{A}'$ with a single state, representing the expected frequencies in a normal word. In this section we refer to the union of $\mathcal{A}$ and $\mathcal{A}'$ as the *union weighted automaton.* More precisely, if $\mathcal{A} = \langle Q, B, \delta, I, F \rangle$ and $\mathcal{A}' = \langle Q', B, \delta', I', F' \rangle$ the states of the union weighted automaton are $Q \cup Q'$, the initial weights are $I \cup I'$, the final weights are $F \cup F'$, and there is a transition $p \xrightarrow{b} q$ of weight w in the union weighted automaton if and only if this transition exists in $\mathcal{A}$ or in $\mathcal{A}'$.

To verify that the frequencies match the expected ones, we assign initial weight -1 to the unique state of $\mathcal{A}'$, so that the union weighted automaton has frequency 0 for all words $w \in B^*$. We make this comparison using Schützenberger's algorithm [**?**]. We refer to the functions realized by $\mathcal{A}$ and $\mathcal{A}'$ as g and g' respectively.

For the output alphabet B, a weighted automaton $\mathcal{A}'$with the expected frequencies in a normal word in B can be described with a single state, having transitions with weight $1/|B|$ for each symbol. To make the comparison between $\mathcal{A}$ and $\mathcal{A}$', we analyze the union automaton, considering $\mathcal{A}'$ with initial weight -1.

Given an arbitrary finite word $w \in B^*$ and a state $q \in Q$, consider the frequency $f_{w,q}$ with which the run labeled with a normal word x visits q and immediately after the remaining output starts with w. We describe a procedure to find a base of the subspace of $\mathbb{R}^{1\times n}$ generated by $\{\pi_w \mid w \in B^*\}$, where $\pi_w = (f_{w,q_1}, \dots, f_{w,q_n})$.

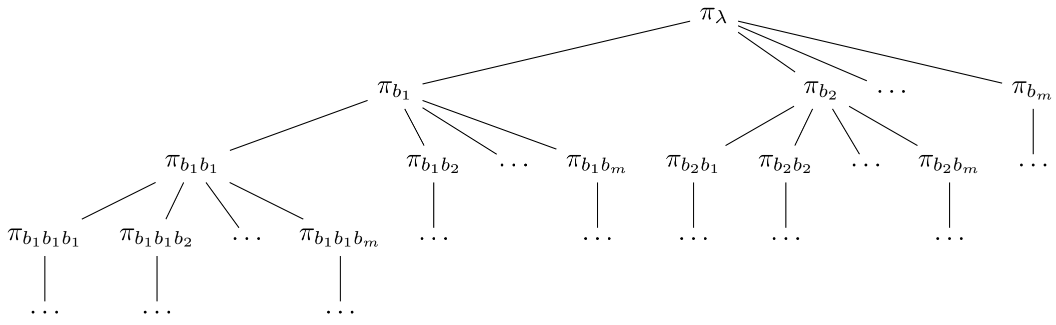
The vector π of initial weights holds information on the frequency with which the output after leaving each state starts with λ when consuming a normal word, so we have that $\pi = \pi_\lambda$. In other words, π holds information on how often a state is visited when consuming a normal word. If $b \in B$, let us define $T(b)$ as the $\mathbb{R}^{n\times n}_{\geq 0}$ matrix such that $(T(b))_{p,q} = \mathit{weight}_{\mathcal{A}}(p \xrightarrow{b} q)$. Then the vector $\pi \cdot T(b)$ holds information on the frequency with which the word b outputs from each state when consuming a normal word.

In general, suppose that the output alphabet is of the form $B = \{b_1, b_2, \dots, b_m\}$. For each word $w \in B^*$, the vector $\pi_w \in \mathbb{R}^{1\times n}$ can be calculated recursively:

$$\begin{cases} \pi_\lambda & = & \pi \\ \pi_{w\cdot b} & = & \pi_w \cdot T(b) \end{cases}$$

For example, $\pi_{b_1b_2b_2} = \pi \cdot T(b_1) \cdot T(b_2) \cdot T(b_2)$.

To calculate a base of the subspace generated by $\{\pi_w \mid w \in B^*\}$, we consider the following infinite tree with branching factor m (one child per each symbol $b \in B$ of the alphabet):

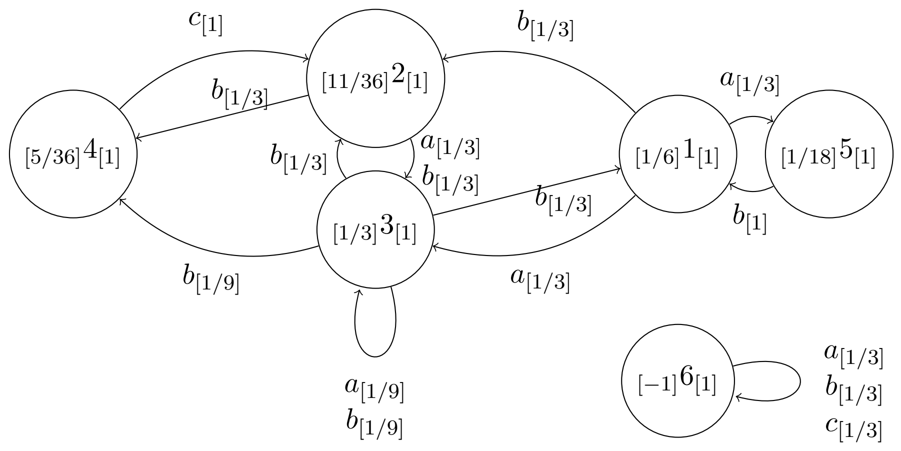


The nodes of the tree are all the vectors π_w, for all finite words $w \in B^*$. The vector π_w is located in the node of the tree following the path indicated by the word w.

We perform a depth-first exploration of the tree, collecting some of the vectors into a linearly independent set $\{v_1, v_2, \dots, v_n\}$. As soon as a node is found to be linearly dependent with the ones collected so far, the subtree is pruned. Note that the subspace is finite-dimensional so this procedure only needs to explore a finite prefix of the infinite tree.

Once the base is calculated, we multiply each of the vectors by $(1\,1\dots 1)^t$, to obtain the frequency of the associated word in the output of a normal word. Since we are analyzing the union weighted automaton (which realizes the difference of functions $g - g'$), we expect this product (frequency) to be zero for every vector. If it is positive, it means the word associated to the vector appears more often than expected (since we are analyzing $g - g'$). If it is negative, the word associated appears fewer times than expected.

Example 20. Below we continue the analysis started on Example 17, building a union weighted transducer, including the original weighted automaton $\mathcal{A}$, and also $\mathcal{A}'$, each with their expected weights. We also exhibit the base of the subspace of frequency vectors, and we perform the verification for each vector in the base.



$$T(a) = \begin{bmatrix} 0 & 0 & 1/3 & 0 & 1/3 & 0 \\ 0 & 0 & 1/3 & 0 & 0 & 0 \\ 0 & 0 & 1/9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/3 \end{bmatrix} \quad T(b) = \begin{bmatrix} 0 & 1/3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 1/3 & 0 & 0 \\ 1/3 & 1/3 & 1/9 & 1/9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/3 \end{bmatrix} \quad T(c) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/3 \end{bmatrix}$$

With the above symbols matrices and the initial weight vector, this is how the base is calculated for the example union weighted automaton:

1. $v_0 = \begin{bmatrix} 1/6 \\ 11/36 \\ 1/3 \\ 5/36 \\ 1/18 \\ -1 \end{bmatrix}$, associated to λ

2. $v_0 T(a)$ is LI with $\{v_0\}$ so $v_1 = \begin{bmatrix} 0 \\ 0 \\ 7/36 \\ 0 \\ 1/18 \\ -1/3 \end{bmatrix}$, associated to a

3. $v_0 T(b)$ is not LI with $\{v_0, v_1\}$ (Hence discarding this branch)

4. $v_0T(c)$ is LI with $\{v_0, v_1\}$, so $v_2 = \begin{bmatrix} 0 \\ 5/36 \\ 0 \\ 0 \\ 0 \\ -1/3 \end{bmatrix}$, associated to c

5. $v_0T(c)T(a)$ is LI with $\{v_0, v_1, v_2\}$, so $v_3 = \begin{bmatrix} 0 \\ 0 \\ 5/108 \\ 0 \\ 0 \\ -1/9 \end{bmatrix}$, associated to ca

6. $v_0T(c)T(b)$ is LI with $\{v_0, v_1, v_2, v_3\}$, so $v_4 = \begin{bmatrix} 0 \\ 0 \\ 5/108 \\ 5/108 \\ 0 \\ -1/9 \end{bmatrix}$, associated to cb.

7. $v_0T(c)T(c)$ is LI with $\{v_0, v_1, v_2, v_3, v_4\}$, so $v_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1/9 \end{bmatrix}$, associated to cc.

8. The rest of the branches to explore result LD.

Thus the vector space base for this example with associated words is:

$$\left\{ \lambda : \begin{bmatrix} 1/6 \\ 11/36 \\ 1/3 \\ 5/36 \\ 1/18 \\ -1 \end{bmatrix}, a : \begin{bmatrix} 0 \\ 0 \\ 7/36 \\ 0 \\ 1/18 \\ -1/3 \end{bmatrix}, c : \begin{bmatrix} 0 \\ 5/36 \\ 0 \\ 0 \\ 0 \\ -1/3 \end{bmatrix}, ca : \begin{bmatrix} 0 \\ 0 \\ 5/108 \\ 0 \\ 0 \\ -1/9 \end{bmatrix}, cb : \begin{bmatrix} 0 \\ 0 \\ 5/108 \\ 5/108 \\ 0 \\ -1/9 \end{bmatrix}, cc : \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1/9 \end{bmatrix} \right\}$$

The next step is to verify that the product of the final weight vector $\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$ and each of the vectors of the base is 0, in other words, verify for each vector of the base that the sum of its coordinates is 0. Since this does not hold for v_5 and we obtain a negative value instead, the algorithm can already return *False* and give some extra information, for a certain normal word the frequency of cc (the word associated to v_5) is too little. It has not been implemented but we can give even more information on what normal word that is, any normal word that leads us to this recurrent strongly connected component in $\mathcal{T}$.

2.2 Correctness

In this section we state the main assertions that prove the algorithm correct.

The following claim shows that it suffices to analyze preservation of normality in each recurrent strongly connected component.

Claim: Each run labeled by a normal word ends in a recurrent strongly connected component.

The following claim shows that the input transducer and its normalized version produce the same output when fed with an infinite word, hence proving it is equivalent to analyze preservation of normality in either of them.

Claim: $\mathcal{T}$ realizes the same function $f : X \subseteq A^\omega \to A^\omega$ before and after normalization.

The following claim ensures that the weighted automaton $\mathcal{A}$ does realize the frequency function for normal inputs.

Claim: For any normal input x, any finite word w, the *frequency of w in y*, where y is the output of $\mathcal{T}$ when fed x, is given by

$$\mathit{freq}(y, w) = \pi \cdot T(a_i) \dots \cdot T(a_n) \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

where $w = b_1 \dots b_n$, the matrix $T(b) \in \mathbb{R}^{n \times n}_{\geq 0}$ is such that $T_{p,q} = \mathit{weight}_{\mathcal{A}}(p \xrightarrow{a} q)$, and π is the stationary distribution of T, the associated stochastic matrix of $\mathcal{A}$.

2.3 Complexity

In this section we analyze the complexity, measured by counting the number of elementary mathematical operations, of a possible implementation of the algorithm. We refer to the size of the input, a deterministic complete transducer $\mathcal{T} = \langle Q, A, B, \delta, I \rangle$, as $n = \sum_{\tau \in \delta} |\tau|$, where the size of a transition $\tau = p \xrightarrow{a|w} q$ is $|\tau| = |aw|$. The analysis is based on an implementation that uses an adjacency matrix to represent transitions. The alphabet is assumed to be fixed.

The following enumerates the main phases of the algorithm and an achievable complexity.

1. Calculate recurrent strongly connected components. The cost of Kosaraju's algorithm is $O(|Q|^2) \subseteq O(n^2)$ if the transducer is implemented with an adjacency matrix [**?**, Section 22.5].
2. Decide if each recurrent strongly connected component $S_1, \dots, S_k$ preserves normality. The size of the component $|S_i|$ is written n_i.
   2.1 Normalize the component. The cost is $O(n_i^2)$, assuming that the transducer is implemented with an adjacency matrix
   2.2 Build the weighted automaton:
      2.2.1 Calculate its transitions and weights. The most expensive step is to calculate E^*. The cost is $O(n_i^3)$ to solve the system of linear equations.

2.2.2 Calculate the initial weights, by calculating the stochastic matrix and solve a system of equations to find its stationary distribution. The cost is $O(n_i^3)$ to solve the systems of equations.

2.2.3 Calculate the final weights. The cost is $O(n_i)$.

2.3 Compare the automaton thus obtained to the one representing the expected frequencies. The cost is $O(|B|n_i^3) = O(n_i^3)$ as B is fixed [**?**].

Overall, the cost of deciding if each component preserves normality is $O(n_i^3)$ for each component, and at most $O(n^3)$ in total, since $\sum_{i=1}^{k} n_i = n$ so $O(\sum_{i=1}^{k} n_i^3) \subseteq O(n^3)$.

Hence, the algorithm has complexity $O(n^3)$ where $n = \sum\limits_{\tau \in \delta} |\tau|$ is the size of the transducer.

3. IMPLEMENTATION NOTES

In this chapter we give an overview of the modules, dependencies and test cases of our prototype implementation. The prototype has been implemented in Python 2.7.

3.1 Modules

We briefly describe the three main classes of our implementation.

3.1.1 `Transducer`

An instance of the `Transducer` class represents a deterministic transducer.

3.1.1.1 Interface

— `class Transducer(states, initialState, inputAlphabet, outputAlphabet)`
  Construct a new deterministic transducer without transitions, where:
  - `states` is the set of states,
  - `initialState` is a single initial state (to enforce determinism),
  - `inputAlphabet` is a set of strings representing the input alphabet,
  - `outputAlphabet` is a set of strings representing the output alphabet.

  The input alphabet should not contain the string `"_"`, as it is reserved to represent λ-transitions by the normalization stage.

— `Transducer.addTransition(p, q, inputSymbol, outputWord)`
  Add a transition to the transducer:
  - `p` is the source state,
  - `q` is the target state,
  - `inputSymbol` is a string which should be among the symbols of the input alphabet,
  - `outputWord` is a list of strings, each of which should be among the symbols of the output alphabet.

— `Transducer.isComplete()`
  Return `True` if and only if the transducer is complete.

— `Transducer.preservesNormality()`
  Return a tuple `(boolean, message)`. The value of `boolean` is `True` if and only if the transducer preserves normality. If the transducer does not preserve normality, `message` is a string containing a human-readable message including a finite word whose frequency is not the expected one in the translation of some normal word.

— `Transducer.toDOT()`
  Return a string that represents the transducer as a graph in DOT format, suitable for rendering it as an image using the Graphviz package[1].

[1] `https://www.graphviz.org`

3.1.1.2 Representation

A deterministic transducer is represented with the following data structure:

```
Transducer._states
Transducer._initialState
Transducer._inputAlphabet
Transducer._outputAlphabet
Transducer._transitions
Transducer._weights
```

where:

- `Transducer._states` is the set of states.
- `Transducer._initialState` is the initial state, which is among the input states.
- `Transducer._inputAlphabet` is a set of strings (the input alphabet).
- `Transducer._outputAlphabet` is a set of strings (the output alphabet).
- `Transducer._transitions` is dictionary of transitions. The keys are of the form `(state, symbol)` and `Transducer._transitions[(p, inputSymbol)]` is of the form `(q, outputWord)`, representing a transition $\mathtt{p} \xrightarrow{\mathtt{inputSymbol|outputWord}} \mathtt{q}$. Remark that the representation enforces determinism.
- `Transducer._weights` is used internally during the normalization stage.

3.1.2 `Automaton`

The `Automaton` class represents a partially built weighted automaton. It has weights on the transitions but it has no initial or final weights on the states.

3.1.2.1 Interface

— `class Automaton(states, initialState, alphabet)`

Construct a new automaton without transitions, where:
- `states` is the set of states,
- `initialState` is a single initial state,
- `alphabet` is a set of strings representing the alphabet.

— `Automaton.addTransitionWithWeight(p, q, inputSymbol, weight)`

Add a transition, where:
- `p` is the source state,
- `q` is the target state,
- `inputSymbol` is a string which should be among the symbols of the alphabet,
- `weight` is the weight of the transition. It should be an instance of any numeric type. Usually it is an instance of the `fractions.Fraction` class of Python's standard library.

— `Automaton.weightedAutomaton()`

Return an instance of `WeightedAutomaton`, representing a weighted automaton. The initial weights are obtained from the stationary distribution of the current automaton (`self`) interpreted as a Markov chain. The final weights are set to 1.

3.1.2.2 Representation

A partially built automaton is represented with the following data structure:

```
Automaton._states
Automaton._initialState
Automaton._alphabet
Automaton._transitions
Automaton._transitionWeights
```

where:

- `Automaton._states` is the set of states,
- `Automaton._initialState` is the initial state,
- `Automaton._alphabet` is a set of strings (the alphabet),
- `Automaton._transitions` is a dictionary of transitions. The keys are of the form `(p, inputSymbol)` and `Automaton.transitions[(p, inputSymbol)]` is a set of states $\{\mathtt{q}_1\texttt{, ..., }\mathtt{q}_n\}$, representing the set of transitions:

$$\left\{\mathtt{p} \xrightarrow{\texttt{inputSymbol}} \mathtt{q}_i \quad : \quad 1 \leq i \leq n\right\}$$

- `Automaton._transitionWeights` is a dictionary associating each transition to its weight. The keys are of the form `(p, inputSymbol, q)` and the value of `Automaton .transitionWeights[(p, inputSymbol, q)]` is an instance of a numeric type, representing the weight of the transition $\mathtt{p} \xrightarrow{\texttt{inputSymbol}} \mathtt{q}$.

3.1.3 `WeightedAutomaton`

The `WeightedAutomaton` class represents a weighted automaton, including weights on the transitions, and also initial and final weights on the states.

3.1.3.1 Interface

— `class WeightedAutomaton(states, initialWeights, finalWeights, alphabet, transitions, transitionWeights)`

Construct a new weigthed automaton:

- `states` is the set of states, which should be consecutive numbers $1, 2, \ldots, n$,
- `initialWeights` is a list of numbers, the i-th element of the list is the initial weight for the i-th state,
- `finalWeights` is a list of numbers, the i-th element of the list is the final weight for the i-th state,
- `alphabet` is a set of strings representing the alphabet,
- `transitions` is a dictionary of transitions, as described in `Automaton`,
- `transitionWeights` is a dictionary of weights, as described in `Automaton`.

— `WeightedAutomaton.preservesNormality()`

Return a tuple `(boolean, message)`. The value of `boolean` is `True` if and only if any two finite words of the same length have the same frequency (which means that the original transducer preserves normality). If the original transducer does not preserve normality, `message` is a string detailing the reason.

3.1.3.2 Representation

A weighted automaton is represented with the following data structure:

```
WeightedAutomaton._states
WeightedAutomaton._initialWeights
WeightedAutomaton._finalWeights
WeightedAutomaton._alphabet
WeightedAutomaton._transitions
WeightedAutomaton._transitionWeights
```

where:

- `WeightedAutomaton._states` is the set of states,
- `WeightedAutomaton._initialWeights` are the initial weights for each state,
- `WeightedAutomaton._finalWeights` are the final weights for each state,
- `WeightedAutomaton._alphabet` is a set of strings (the alphabet),
- `WeightedAutomaton._transitions` is the dictionary of transitions, as described in `Automaton`,
- `WeightedAutomaton._transitionWeights` is the dictionary of weights, as described in `Automaton`.

3.2 Dependencies

Our prototype implementation is written in Python 2.7 and has been tested using the official Python distribution (CPython).

For dealing with matrices and systems of linear equations, we use the SymPy 1.1 package[2]. SymPy is a Python library for symbolic computation. In particular, we rely on SymPy to solve the systems of linear equations which define the matrix E^*, required to build the weigthed automaton, and to calculate the stationary distribution which defines the initial weights for all states. Moreover, to compare the weighted automaton with the expected frequencies, we also need to test whether a vector is linearly independent to a set of vectors. For this step we also rely on SymPy.

Initially, we considered NumPy[3] for dealing with matrices. However, NumPy is a library oriented to numerical computation, and solutions are expressed as floating point numbers, whereas in our case we need to obtain exact solutions to systems of linear equations involving matrices with rational coefficients.

3.2.1 Graphics

As mentioned before, we represent transducers and automata in DOT format to be able to render them graphically using the Graphviz package[4].

3.3 Test cases

In this section we exhibit some of the cases with which we have tested the algorithm. Test cases are classified according to whether the transducers involved preserve normality

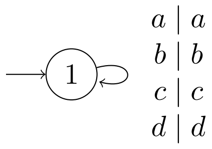
[2] http://www.sympy.org/en/
[3] http://www.numpy.org/
[4] http://www.graphviz.org

(Section 3.3.1) or not (Section 3.3.2).

3.3.1 Transducers that preserve normality

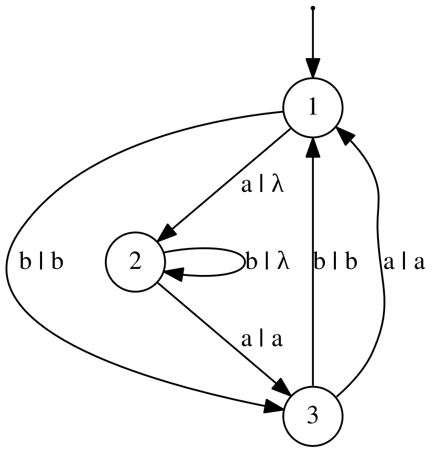
In this section we present test cases of transducers that preserve normality. We generate them using known examples of transducers that preserve normality, such as Agafonov selectors, deleting all occurences of a symbol from the input, changing only finite symbols from the input.

Test case #1.



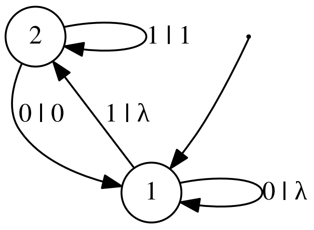
Preserves normality since it is the identity function (also it is a case of Agafonov's selector).

Test case #2.



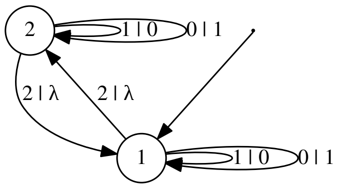
Preserves normality since it behaves just as the Agafonov transducer resulting from changing 1 $\xrightarrow{a|\lambda}$ 2 for 1 $\xrightarrow{a|a}$ 2 and 2 $\xrightarrow{a|a}$ 3 for 2 $\xrightarrow{a|\lambda}$ 3.

Test case #3.



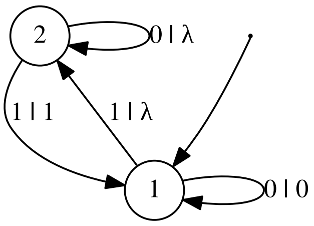
Preserves normality since it is an Agafonov's selector.

Test case #4.



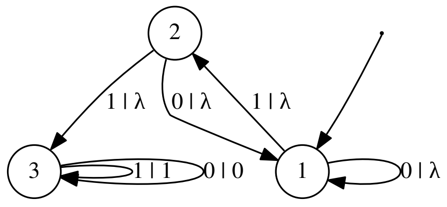
Preserves normality since it removes all occurrences of symbol 2 and permute the others.

Test case #5.



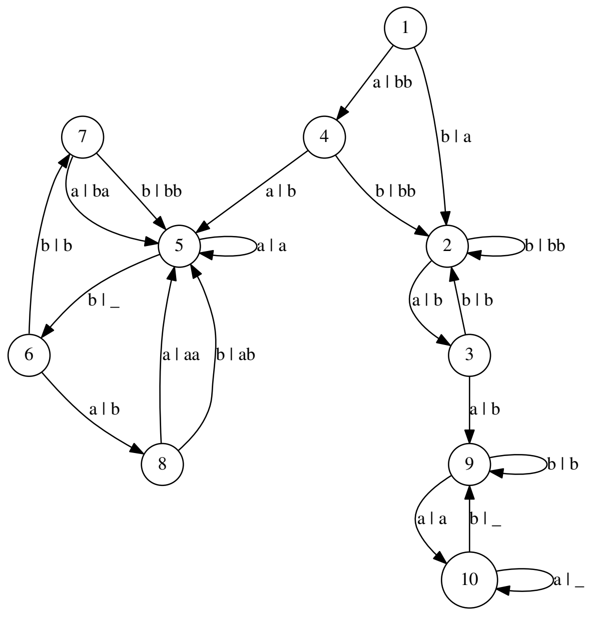
Preserves normality since it behaves just as the agafonov transducer resulting from changing 1 $\xrightarrow{1|\lambda}$ 2 for 1 $\xrightarrow{1|1}$ 2 and 2 $\xrightarrow{1|1}$ 1 for 2 $\xrightarrow{1|\lambda}$ 1.

Test case #6.



Preserves normality as it only deletes finite symbols from a normal word (Deletes every symbol until the first appearance of 11).

Test case #7.

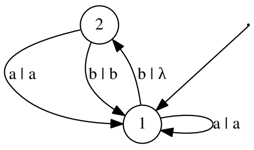


Preserves normality since each of its recurrent strongly connected components do.

3.3.2 Transducers that do not preserve normality

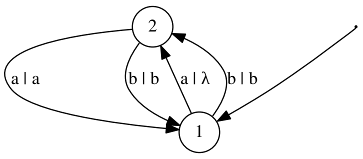
In this section we show test cases that do not preserve normality, we generate them by forcing certain finite words to have the wrong frequency. In each there is a brief explanation of why it does not preserve normality.

Test case #8.



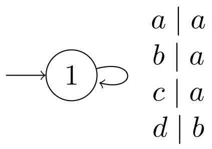
For any normal sequence, the output is not normal as the frequency of ***a*** is larger than the frequency of ***b***

Test case #9.



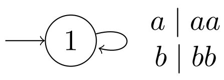
For any normal sequence, the output is not normal as the frequency of ***b*** is larger than the frequency of ***a***

Test case #10.



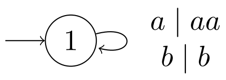
For any normal sequence, the output is not normal as the frequency of ***a*** is larger than the frequency of ***b***

Test case #11.



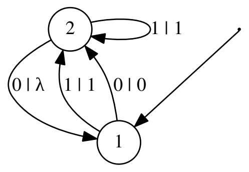
For any normal sequence, the output is not normal as the frequency of ***aaa*** is larger than the frequency of ***aba*** (which is 0 in this case)

Test case #12.



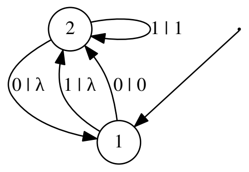
For any normal sequence, the output is not normal as the frequency of ***aaaa*** is larger than the frequency of ***abab***

Test case #13.



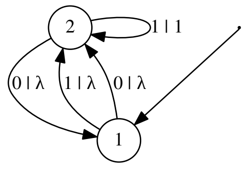
For any normal sequence, the output is not normal as the frequency of ***1*** is larger than the frequency of ***0***

Test case #14.



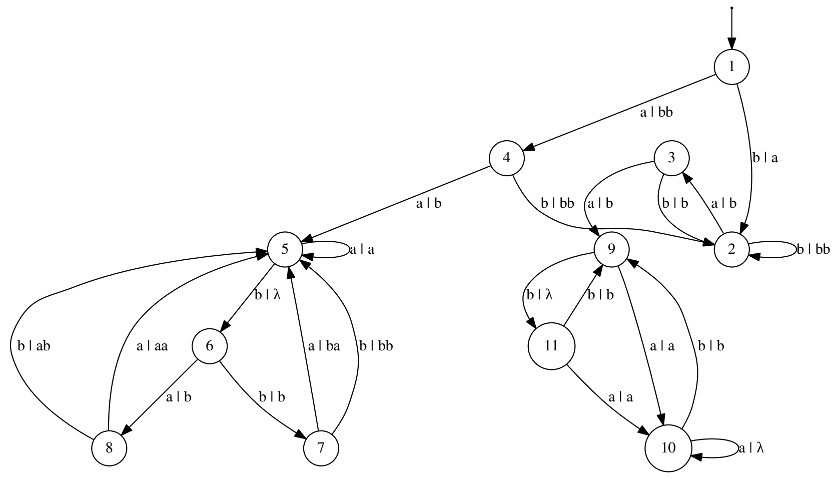
For any normal sequence, the output is not normal as the frequency of ***1*** is larger than the frequency of ***0*** (Notice that blocks of ***0***s only output half its ***0***s, while blocks of ***1***s may at most output one fewer ***1***)

Test case #15.



For any normal sequence, the output is not normal as no ***0***s are displayed.

Test case #16.



This transducer does not preserve normality as any normal word beginning with *bab* leads to recurrent strongly connected component $\{9, 10, 11\}$, that does not preserve normality since the frequency of b is smaller than the frequency of a in the output (Blocks of as output just one a, while blocks of bs output half the number of bs in the block).

4. CONCLUSION

In the present thesis we describe an algorithm to determine preservation of normality for a given deterministic transducer. It remains to present a full proof of correctness.

Additionally, the implemented prototype gives an incomplete witness when the transducer does not preserve normality. The algorithm currently returns a word that has the wrong frequency in some normal word, but it does not specify which normal word that is. This feature could be added by showing the path that leads to a failing recurrent strongly connected component, as the output of a normal word with such path as prefix will not be normal. The structures used could be improved to achieve better efficiency. Also, the features to solve systems of equations could be implemented to benefit from the fact that we only deal with matrices that have fractional numbers as coefficients, instead of using a general library as the SymPy package.

A different yet related question appears: given the desired frequency of each letter in the output alphabet, can we build a transducer that outputs a word with such properties when being supplied with a normal word? This is an open question.