國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

加權有限自動機最小化

On Minimizing Weighted Finite Automata

黃亭遠

Ting-Yuan Huang

指導教授: 顏嗣鈞　教授

Advisor: Hsu-Chun Yen, Professor

中華民國九十七年七月

July, 2008

誌謝

這篇論文的完成是我人生中一個重要的里程碑, 也是台大兩年研究所生活的總結. 在這些日子裡, 有許多人默默地幫助我, 支持我. 在此, 我想特別對他們表達我的敬意與謝意.

給我的父母, 感謝您們的養育與栽培. 給我的指導教授顏嗣鈞老師, 感謝您的指導與協助. 給郭斯彥教授, 雷欽隆教授, 莊仁輝教授, 黃秋煌教授, 感謝您們於百忙之中撥冗指正. 給來自 Ecole nationale supérieure des télécommunications 的 Jacques Sakarovitch 教授, 來自 Université Paris-Est Marne-la-Vallée 的 Sylvain Lombardy 教授, 來自 Ecole d'Ingénieurs en Informatique 的 Michäel Cadilhac 先生, 感謝你們的建議及幫助. 給我的女友, 感謝妳的支持與鼓勵. 給我的同學們, 感謝你們的照應與陪伴. 謝謝你們!

黃亭遠　謹識

于　計算理論研究室

中華民國九十七年七月

摘要

加權自動機是一種可以用來模塑許多系統, 非常廣泛的數學模型. 在不同的研究領域裡它有不一樣的名字, 例如機率有限自動機 (probabilistic finite automata, PFA) 和隱藏式馬可夫模型 (hidden Markov model, HMM) 等等. 這些模型在很多研究領域裡皆具極爲重要的地位, 例如機器學習 (machine learning), 語音處理 (speech processing), 圖像辨認 (pattern recognition), 語言塑模 (language modeling), 生物資訊 (bioinformatics), 電路測試 (circuit testing), 影像處理 (image processing) 及時間序列分析 (time series analysis). 因爲加權自動機有著非常多的應用領域, 故只要有些微的理論突破或改進, 影響便非常深遠. 在本篇論文中, 藉由探討加權自動機中權重的代數性質及其拓撲結構, 我們引進了重新分佈權重的概念. 我們並將最短路徑問題 (shortest-path problem) 加以推廣, 使其定義於各路徑之最大下界 (infimum) 之上, 同時給出一有效之演算法解決之. 我們提出之演算法不僅是本篇論文中縮小自動機的方法核心, 也能應用於辨別整數自動機 (Z-automata) 的等價性. 我們的方法能有效的縮小自動機, 對於之前文獻中的方法有十分顯著的改進.

Abstract

Weighted Finite Automata represent a very general model which has many different names such as probabilistic finite automata (PFA), hidden Markov models (HMM), stochastic regular grammars, Markov chains and n-grams. These models play central roles in many domains such as machine learning, speech processing, computational linguistics, pattern recognition, language modeling, bioinformatics, music modeling, circuit testing, image processing, path query and time series analysis. The huge number of applications makes weighted finite automata a very valuable research topic: Even a very small breakthrough or improvement would benefit lots of domains. In this thesis, we introduce the notion of "weight redistribution" by investigating the algebraic properties along with the graphical structure inside weighted finite automata. We also generalize the concept of shortest-path problem to finding infimum along every paths and give an effcient algorithm to solve it. Our algorithm to compute "weight redistribution" not only plays the central role in our minimization algorithm, but also is applicable on determining the equivalence Z-automata. We also give two new algorithms to shrink the state space. These algorithms outperform pervious results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Weighted finite automata is a generalization to the standard finite automata by associating weights on transitions. The weights can be defined over every semirings and hence a weighted finite automaton maps strings to some semiring. Note that a standard finite automaton usually maps strings to $\{0, 1\}$.

Weighted finite automata have many specializations such as probabilistic finite automata (PFA) [2], hidden Markov models (HMM) [3], stochastic regular grammars [4], Markov chains [5] and n-grams [6]. Although names and definitions are varying, they are indeed equivalent to or special cases of weighted finite automata. These models are very important and are central topics in corresponding domains such as machine learning, speech processing, computational linguistics, pattern recognition, language modeling, bioinformatics, music modeling, circuit testing, image processing, path query, time series analysis and many more [1]. The huge number of applications makes weighted finite automata a very valuable research topic: Even a very small breakthrough or improvement would benefit lots of domains.

Among several main topics on weighted finite automata like augmenting the expressive power, determinizations or reducing nondeterminisms, one is on how to reducing the size of weighted finite automata. In many applications the state space can grow upto 10^{10} and make these applications impractical. So it is very important to shrinking the size of weighted finite automata while retaining some kinds of behaviors. One of the most common and practical objective function is the state space. Although sometimes we may more care about transitions, in most situations the state space is of main concern.

It is usually possible to find the minimal form of a deterministic (weighted) finite automaton efficiently [8, 7] while in most cases of nondeterministic (weighted) finite automata, finding a minimal form, that is, an equivalent automaton with minimum states, is P-SPACE hard [9]. Even that a unique minimal form can not be assured.

There are many researchs of minimizing standard deterministic finite automata [8, 7, 10], deterministic weighted finite automata [11] and deterministic finite transducers [12, 13]. Almost all of the algorithms regarding to these deterministic automata run in $O(N^3)$ while others have their speciality like Brozozowski's algorithm which requires exponential time.

The best known result of minimizing deterministic finite automata is perhaps Hopcroft's $O(N \lg N)$ algorithm. Like most of the other algorithms, it utilizes the partition refinement technique. Although it can be applied to nondeterministic finite automata to help shrinking the state space while preserving the same language, the minimality can not be retained.

Mohri's method generalizes the idea of the partition refinement technique from

standard deterministic finite automata to deterministic weighted finite automata defined over tropical semirings and finite transducers [12]. In Mohri's algorithm the weights are treated as labels. In his works the researchs and results are mostly about deterministic automata. The nondeterministic ones are seldom touched.

In Mohri's algoirthm, there are some constraints to the semiring residing in the weighted finite automata. Eisner generalized and formalized the requirements of minimizing deterministic weighted finite automata and showed that the unique minimal form exists only if a greedy factorization of the underlying semiring is possible [11]. In a few words, a division semiring, namely a skew field, is needed by Eisner's algorithm. To be more precise, the requirements are put on the mulplicative monoids inside the semirings. Though that Eisner's algorithm is applicable to deterministic weighted finite automata over any semiring, the minimality can not be assured without those constraints.

K-Quotient is another notion to shrink the state space of weighted finite automata and is applicable on all semirings. It is also a generalization to Hopcroft's algorithm. It makes different uses of the algebraic structure underlying the weighted finite automata to Mohri's algorithm as explained in Chapter 4.

Due to the hardness of finding minimal forms of nondeterministic finite automata, many researchs focused on the heuristics of shrinking the state space [14]. Most of them exploit the behavioral similarities among states to merge redundant states. Among the most famous methods are bisimulation minimization [14, 15] and simulation minimization [14, 17].

The weighted cases are slightly different. In fact, the minimal form of Z-

automata, i.e. the weighted finite automata over semiring $(Z, +, *, 0, 1)$, can be found in polynomial time. But given a weighted finite automata over tropical semiring, that is $(R \cup \infty, min, +, \infty, 0)$, also known as min-plus algebra, the problem is undecidable [16]. In general, the problem on finding minimal forms is hard so we must consider some heuristic approaches.

Bisimulation relation can also be defined over weighted finite automata [15]. The simulation relation, on the other hand, can be ambiguous over weighted finite automata because the computation of weighted finite automata consists of summing up weights associated by every paths rather than just consider the reachability as in the case of standard finite automata. The best results on minimizing general weighted finite automata to our knowledge is based on bisimulation minimization.

In this work, we introduce the notion of "weight redistribution" by investigating the algebraic properties along with the graphical structures inside weighted finite automata. We generalize the concept of shortest-path problem to finding infimum along every paths and give an effcient algorithm to solve it. Our algorithm to compute "weight redistribution" not only plays the central role in our minimization algorithm, but also is applicable on determining the equivalence Z-automata. We give two new algorithms to shrink the state space. These algorithms outperform pervious results. Our algorithms is also a generalization of the algorithm in [12] which can minimize deterministic finite transducers because finite transducers are special cases of weighted finite automata as explained in the next chapter. Although Eisner generalized Mohri's works about tropical semirings in [13] to that in [11] which is not a special case of ours, our algorithms are applicable on quite different

Table 1.1: Comparison of different minimization algorithms.

algorithms	applicable domains
Mohri's	deterministic finite transducers
	weighted finite automata over tropical semirings
Eisner's	deterministic weighted finite automata over division semirings
K-Quotient	weighted finite automata over all kinds of semirings
Ours	weighted finite automata with $\otimes$ cancellative and pure

kind of semirings to Eisner's. The comparison of our work and previous works is summarized in table 1.1.

In Chapter 2, we review some definitions on finite automata, weighted finite automata and finite transducers.

In Chapter 3, we firstly set up the algebraic structure needed by our study then introduce the concept of "weight pushing". A weight-pushing based algorithm for computing "weight redistribution" is given for illustrating the concepts of weight redistribution and for comparison. This algorithm is powerful and not too slow in usual cases. An efficient algorithm is presented in the end of this chapter.

In Chapter 4, we study the notion of Nerode quotient to minimize the weighted finite autoata. We describe the structure which can be exploited to further shrink the state space and combine the results in Chapter 3 to deliver an efficient and powerful minimization algorithm. We conclude that our algorithm can be applied in the other direction of transitions at the end of chapter 4.

In Chapter 5, we give some experimental results to show that our algorithms are

much better.

In Chapter 6, we conclude our works and mention several possible future works.

Chapter 2

Preliminaries

2.1 Finite Automata

Let Σ be a finite alphabet and Σ^* the set of all strings that can be obtained from Σ including the empty string ϵ. A language is a subset of Σ^*.

By convention, symbols in Σ are denoted by letters from the beginning of the alphabet $\{a, b, c, ...\}$ and strings in Σ^* are denoted by the end of the alphabet letters $\{..., x, y, z\}$. The length of a string $x \in \Sigma^*$ is written in $|x|$. The set of all strings of length n is denoted by Σ^n.

Definition 1. *A deterministic finite automaton D, abbreviated as DFA, is a five-tuple (Q, Σ, T, q_0, F):*

- *Q is a finite set of states.*
- *Σ is the input alphabet.*
- *$T \in Q \times \Sigma \times Q$ is the transition function.*

- q_0 *is the initial state.*

- $F \subset Q$ *is the set of final states.*

A transition $t = (p, a, q) \in T$ can be interpreted as that on input a the current state goes from p to q. We write $src(t) := p, dst(t) := q, label(t) := a$, given $t = (p, a, q)$.

T is often defined equivalently:

- $\delta : Q \times \Sigma \to Q$ is the transition function.

where $\delta(p, a) = q$ means that, on input a the current state can go from p to q. So given $p, q \in Q$, $a \in \Sigma$, $(p, a, q) \in T$ if and only if $\delta(p, a) = q$. When the transition function δ is total, the automaton is said to be complete.

The extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ is recursively defined in this way:

$$\begin{cases} \hat{\delta}(q, \epsilon) = q \\ \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a) \end{cases}$$

So $\hat{\delta}(q, w) = p$ means that given q as current state, after reading w, it goes to p.

A state $p \in Q$ is said to be accessible if $\exists w \in \Sigma^* \Rightarrow \hat{\delta}(q_0, w) = p$. A state $p \in Q$ is said to be co-accessible if $\exists w \in \Sigma^*, f \in F \to \hat{\delta}(p, w) = f$.

The language $L(D)$ accepted by D is the set of all strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w) \in F$. Two DFA D and D' are equivalent if and only if $L(D) = L(D')$. A DFA is called minimal if there is no other equivalent DFA with fewer states. States $p, q \in Q$ are said to be equivalent, denoted as $p \approx q$, if and only if $\forall w \in \Sigma^*, \hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$. The equivalent automaton $D/\approx$ is called quotient of D. States

in $D/\approx$ are correspond to the equivalence classes induced by $\approx$. $D/\approx$ is proved minimal and unique up to graph isomorphism.

A deterministic finite automaton D can be seen as a function that $D : strings \to \{0, 1\}$.

Definition 2. *A nondeterministic finite automaton N, abbreviated as NFA, is a five-tuple (Q, Σ, T, I, F):*

- *Q is a finite set of states.*
- *Σ is the input alphabet.*
- *$T \in Q \times \Sigma \times Q$ is the transition function.*
- *$I \subset Q$ is the set of initial states.*
- *$F \subset Q$ is the set of final states.*

A transition $t = (p, a, q) \in T$ can be interpreted as that on input a the current state goes from p to q. We write $src(t) := p, dst(t) := q, label(t) := a$, given $t = (p, a, q)$.

Since N is nondeterministic, T is often defined equivalently:

- $\delta : 2^Q \times \Sigma \to 2^Q$ is the transition function.

where $q \in \delta(p, a)$ means that on input a the current state can go from p to q. So given $p, q \in Q$, $a \in \Sigma$, $(p, a, q) \in T$ if and only if $q \in \delta(p, a)$.

The extended transition function $\hat{\delta} : 2^Q \times \Sigma^* \to 2^Q$ is recursively defined in this way:

$$\begin{cases} \hat{\delta}(q, \epsilon) = q \\ \hat{\delta}(S, xa) = \delta(\hat{\delta}(S, x), a) \end{cases}$$

The language $L(N)$ accepted by N is the set of all strings $w \in \Sigma^*$ such that $\hat{\delta}(I, w) \cap F \neq \emptyset$. Two DFA N and N' are equivalent if and only if $L(N) = L(N')$. A NFA is called minimal if there is no other equivalent NFA with fewer states. States $p, q \in Q$ are said to be equivalent, denoted as $p \approx q$, if and only if $\forall w \in \Sigma^*$, $r \in \hat{\delta}(p, w), r \in F \Rightarrow \exists s \in \hat{\delta}(q, w), s \in F$ and $s \in \hat{\delta}(q, w), s \in F \Rightarrow \exists r \in \hat{\delta}(p, w), r \in F$. The equivalent automaton $D/\approx$ is called quotient of N. States in $N/\approx$ are correspond to the equivalence classes induced by $\approx$. $D/\approx$ is not necessary minimal and unique up to graph isomorphism.

All language accpeted by a NFA can also be accepted by a DFA. A language is regular if and only if it can be accpeted by a DFA or NFA.

A nondeterministic finite automaton N can be seen as a function that N : $strings \rightarrow \{0, 1\}$.

2.2 Weighted Finite Automata

Definition 3. *A semiring $(K, \oplus, \otimes, \bar{0}, \bar{1})$ is a set K with binary operations $\oplus$ and $\otimes$ defined over K such that the following axioms are satisfied:*

- $\oplus, \otimes$ *are associative.*
- $\oplus$ *is commutative.*
- $\otimes$ *distributes over* $\oplus$.
- $\bar{0}$ *is the additive identity.*
- $\bar{1}$ *is the multiplicative identity.*

- $\bar{0} \neq \bar{1}$.

- $\forall k \in K, k \otimes \bar{0} = \bar{0} \otimes k = \bar{0}$.

For example, $(N, +, *, 0, 1)$ and $(R \cup \infty, min, +, \infty, 0)$ are semirings.

Since the deterministic automata are special cases of nondeterministic ones, the automata we defined in the sequel are assumed to be nondeterministic unless specified.

Definition 4. *A weighted finite automaton M, abbreviated as WFA, is a six-tuple $(K, Q, \Sigma, T, initial, final)$:*

- *Weights are defined over K.*

- *Q is a finite set of states.*

- *Σ is the input alphabet.*

- *$T \in Q \times \Sigma \times K \times Q$ is the transition function.*

- *$initial : Q \to K$ specifying the initial weights.*

- *$final : Q \to K$ specifying the final weights.*

A transition $t = (p, q, a, w) \in T$ can be interpreted as that on input a the current state goes from p to q while emitting weight w. We write $weight(t) := w, src(t) := p, dst(t) := q, label(t) := a$ given $t = (p, q, a, w)$.

A path π in M is a sequence of consecutive transitions $t_1 t_2 ... t_n$ with $dst(t_i) = src(t_{i+1})$ for $i = 1, 2, ..., n-1$. We define the label of π, denoted by $label(\pi)$, by

$label(\pi) := label(t_1)label(t_2)...label(t_n)$. The weight associated by π, denoted by $weight(\pi)$, is defined by:

$$weight(\pi) := initial(src(t_1)) \otimes weight(t_1) \otimes weight(t_2) \otimes ... \otimes weight(t_n) \otimes final(dst(t_n)).$$

The weight associated by a string x is defined by:

$$weight(x) := \bigoplus_{\pi : label(\pi) = x} weight(\pi).$$

A weighted finite automaton M can be seen as a function that $M : strings \rightarrow K$. The weighted finite automaton in figure 2.1 computes the numerical values of the binary representations if we consider b as 1 and a as 0. For example, a input string "babb" can be parsed by following ways:

$$\begin{aligned} \texttt{babb} &\Rightarrow 1 \times 2 \times 2 \times 2 = 8 \\ \texttt{babb} &\Rightarrow 1 \times 1 \times 1 \times 2 = 2 \\ \texttt{babb} &\Rightarrow 1 \times 1 \times 1 \times 1 = 1 \\ \hline \texttt{babb} &\Rightarrow 11 \\ \texttt{0b1011} &= 11 \end{aligned}$$

We can get the value of "babb" by summing up every values associated by every pathes.

Two weighted finite automata are equivalent if they associate the same weight to each input string. The distribution of the weights among paths need not to be the same; Only the $\oplus$ sum is required to be the same.

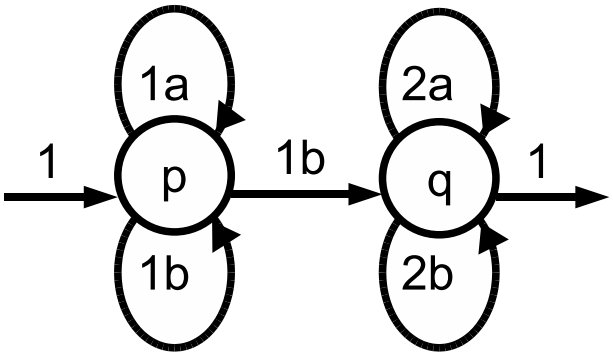


Figure 2.1: A WFA computing numerical values of binary representations.

2.3 Finite State Transducer

Definition 5. *A nondeterministic finite transducer X, abbreviated as NFT, is a six-tuple $(Q, \Sigma, \Delta, T, I, F)$:*

- *Q is a finite set of states.*
- *Σ is the input alphabet.*
- *Δ is the output alphabet.*
- *$T \in Q \times \Sigma \times \Delta \times Q$ is the transition function.*
- *$I \subset Q$ is the set of initial states.*
- *$F \subset Q$ is the set of final states.*

A transition $t = (p, a, b, q) \in T$ can be interpreted as that on input a the current state goes from p to q while outputing b. We write $src(t) := p, dst(t) := q, label(t) := a, output(t) := b$, given $t = (p, a, b, q)$.

A finite transducer X can be seen as a function that $X : strings \rightarrow strings$.

In fact, a finite transducer X is also a weight finite automata M. With semiring $(strings, union, concatenation, \emptyset, \epsilon)$, we can regard Δ of T_X of X as K of T_M of M.

Chapter 3

Weight Redistribution

Weight redistribution is an operation which is language preserving, or equally, realize the same series on weighted finite automata and finite transducers. It only redistribute weights to edges; Neither it modifies the structure of the underlying directed graph nor alphabetical labels on transitions. After applying weight redistribution, the final weights and the weights of outgoing transitions are pushed into initial weights and the weights of incoming transitions as much as possible.

Our weight redistribution algorithm is not only useful on minimizing weighted automata in the next chapter but also effective on determining the equivalence of deterministic transducers [12] and equivalence of z-automata [18]. In fact, the algorithm proposed in [12] is a special case of our weight redistribution algorithm. Our algorithm can be applied on automata with weights in semirings on which the greatest common divisors are properly defined. These semirings are, to just name a few, $(Z^+, *, +, 1, 0), (N, +, min, \infty, 1)$ and $(A^*, concatenation, union, \emptyset, \epsilon)$.

In this chapter, we'll firstly set up the requirements of the underlying algebraic

structure. The requirements are different to all the previous works. That is, we are dealing with different weight types compared to previous works. In fact, the targets of our study are weighted finite automata whose weights are over semirings on which the greatest common divisors are well defined.

After setting up the neccessary mathematical requirements, we'll introduce weight pushing on individual states. Then, we give a weight-pushing based algorithm which redistributes weights by iteratively apply weight pushing on all states in an automaton. We claim that this weight-pushing based algorithm will eventually halts by showing that the automaton finally goes stable.

The weight-pushing based algorithm performs quite well although in some cases it can be very inefficient. We propose an efficient algorithm which reduces the problem into a generalized shortest-path problem and computes the stablized structure directly. Also this algorithm is better than the weight-pushing based algorithm in terms of state space when they are used in next chapter.

3.1 Requirements of the Algebraic Structure

The targeting automata are weighted finite automata whose weights are over semirings on which the greatest common divisors are well defined. To be more precise, the requirements are put on the $\otimes$ operators, that is, on the monoids which is the one that is not neccessary communicative inside the semirings.

In this section we'll first review some necessary materials in abstract algebra, then to describe formally the requirements on the targeting automata.

Let's state a few well known definitions and propertes without proofs. We assume

that a monoid M denoted as $(M, \otimes, \bar{1})$ is given.

Proposition 1. *$\forall a \in M$, there is at most one solution x to*

$$a \otimes x = x \otimes a = \bar{1}$$

where $x \in M$.

Definition 6. *$a \in M$ is invertible if there exist x such that $a \otimes x = x \otimes a = \bar{1}$. We denote x by a^{-1} and call it the inverse of a. The set of all invertible elements in M is denoted by $Inv(M)$.*

Definition 7. *M is a pure monoid if and only if $Inv(M) = \{\bar{1}\}$*

That is, none of the elements in M is invertible except $\bar{1}$. For example, $(N, *, 1)$ and $(N, +, 0)$ are pure but $(Z, +, 0$ and $(Q, *, 1)$ are not.

Definition 8. *Given $a, b \in M$. a is a divisor of b and b is a multiple of a if*

$$a \otimes x = b$$

for some $x \in M$, and we write a div b. We define operator $\oslash$ by

$$b \oslash x = a$$

Definition 9. *For a given subset S of M, we define the set of all common divisors of S by*

$$cd\_of(S) = \{a \in M | \forall b \in S, a \ div \ b\}$$

and the set of all common multiples of S by

$$cm\_of(S) = \{b \in M | \forall a \in S, a \ div \ b\}$$

By this definition, obviously $\bar{1} \in cd\_of(S)$, $\forall S \subset M$.

Proposition 2. *The relation div in M is reflexive and transitive. If M is cancellative and pure then div is also antisymmetric and hence a partial order.*

Definition 10. *Assume that M is cancellative and pure, so that div is an order in M. $p \in M$ is called a prime element if*

$$\forall a \in M, a \ div \ p \Rightarrow a = \bar{1} \ or \ a = p.$$

In other words, a prime element p is a minimal element of $M \backslash \{\bar{1}\}$ relative to div. The set of all prime elements of M is denoted by $Prime(M)$.

Now we are prepared to define the greatest common divisor and smallest common multiple.

Definition 11. *Let M a cancellative and pure monoid and $S \subset M$. If S has an infimum relative to div, we call this infimum the greatest common divisor of S and denote it by $gcd\_of(S)$. If S has an supremum relative to div, we call this supreimum the smallest common multiple of S and denote it by $lcm\_of(S)$.*

Or, equivalently, we can put the constraints on a semiring on which the weights in a weighted automaton defined as following.

Corollary 1. *In semiring $(K, \otimes, \oplus, \bar{1}, \bar{0})$, The greatest common divisors and smallest common multiple can be defined over $\otimes$ if and only if $(K, \otimes, \bar{1})$ is pure and cancellative.*

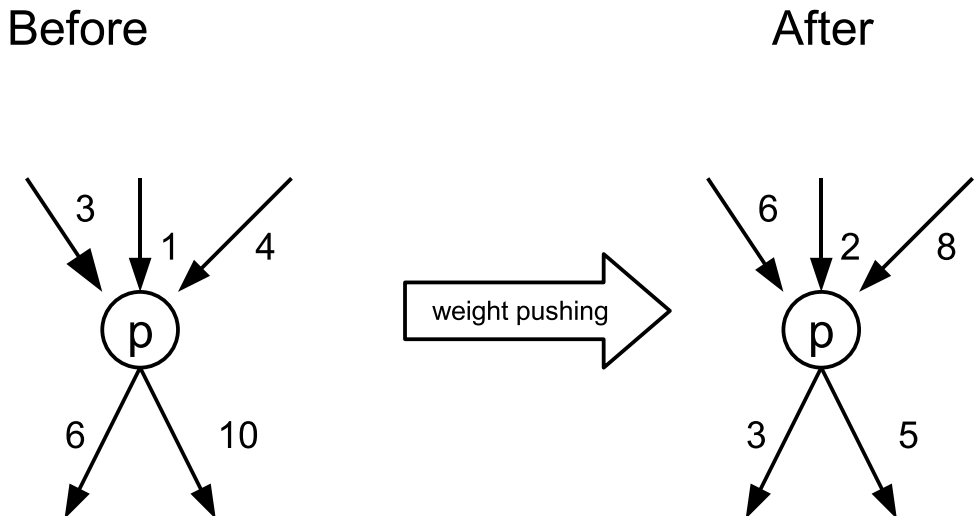


Figure 3.1: An example of weight pushing on $(N, *, +, 1, 0)$.

3.2 Weight Pushing

Weight pushing is an operation which intuitively pushes the weights of some state's outgoing transitions into its incoming transitions. Less formally, on state p we push weights of its outgoing transitions into its incoming transitions by dividing (the weights of) all outgoing transitions by some of their common divisors d and then multiply all the incoming edges by d. We assume d is the greatest common divisor in the sequel.

In figure 3.1, for example, the weights of p's outgoing transitions are pushed into incoming transitions. In this example the weights of transitions are in $(N, *, +, 1, 0)$.

After introducing weight pushing, we are ready to give a weight-pushing based algorithm for redistributing weights from final weights into initial weights. Without loss of generality, we assume the automata are all trimed, namely, all of the states

are accessible and co-accessible.

As in algorithm WEIGHT-REDISTRIBUTION-BY-PUSHING, weights of transitions of each state are iteratively pushed until the structure becomes stable. Note that we consider initial weights and final weights as incoming and outgoing transitions respectively and just ignore self-loops since it won't be modified during weight pushing.

Although it is not obvious, algorithm WEIGHT-REDISTRIBUTION-BY-PUSHING always halts.

Theorem 1. *Algorithm* WEIGHT-REDISTRIBUTION-BY-PUSHING *always halts.*

Proof. Since there are no isolated states, that is, all states are accessable and co-accessable by assumption, each weight pushing operation eventually consumes some of the final weights. There exist no chains of weight pushing which consume no final weights unless some states in the chain are not reaching final states that is ruled out by our assumption.

Also that the final weights are finite, eventually they are run out and algorithm WEIGHT-REDISTRIBUTION-BY-PUSHING halts. □

Though that this weight-pushing based algorithm works, it is indeed very inefficient. There are cases that make this weight-pushing based algorithm very slow. In figure 3.3, there can be only 2 to push in each pass because the weight of transition from p to q is only 2, hence in this example it requires n passes to go stable.

Algorithm WEIGHTS-REDISTRIBUTION-BY-PUSHING

1: **repeat**
2: **for all** states p **do**
3: $gcd \leftarrow 0$
4: **for all** weight w of outgoing transition t of p **do**
5: $gcd \leftarrow gcd\_of(gcd, w)$
6: **end for**
7: **for all** weight w of outgoing transition t of p **do**
8: $w \leftarrow w \oslash gcd$
9: **end for**
10: **for all** weight w of incoming transition t of p **do**
11: $w \leftarrow w \otimes gcd$
12: **end for**
13: **end for**
14: **until** there are no states to push any more.

Figure 3.2: A weight-pushing based weight redistribution algorithm.

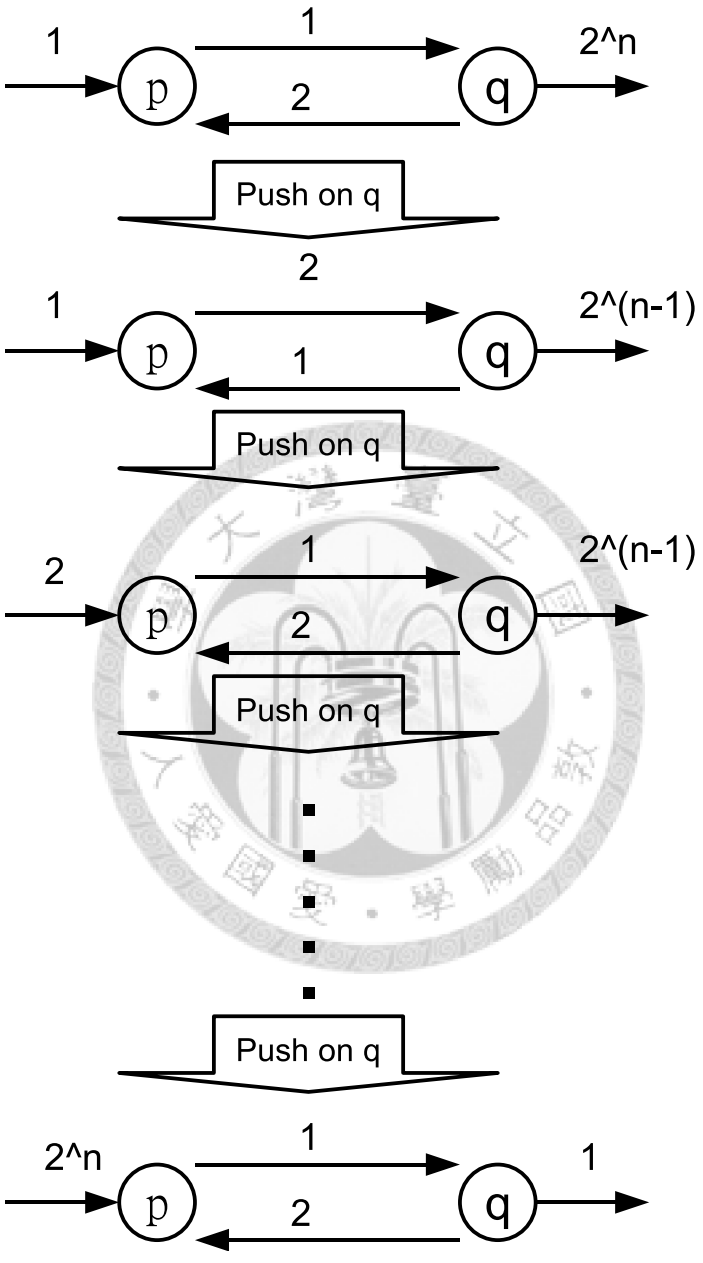


Figure 3.3: A bad case to the weight-pushing based algorithm.

3.3 An Efficient Algorithm

As the readers may notice, in the weight-pushing based algorithm, every transitions with the same destinations are multiplied by the same set of elements. Symmetrically, every transitions with the same sources are divided by the same set of elements. Instead of weight pushing on each single state, the weight redistribution can thus be rediscribed by assigning each state a potential weight then multiply and divide each transition by the potential weight of its destination and source respectively. There's no difference between redistributing weights of an automaton by the weight-pushing based algorithm and by multiplying and dividing by the corresponding potential weights.

To realize the concept, algorithm WEIGHT-REDISTRIBUTION-EFFICIENTLY redistributes weights by changing the weights of each transition directly. It relies on the potential function $d()$ to calculate the new weights.

Now the problem is how we get $d()$. There is clearly a candidate to compute the potential function $d()$: algorithm WEIGHT-REDISTRIBUTION-BY-PUSHING During each iteration of the weight-pushing based algorithm, the greatest common divisor of the weights of the outgoing transitions of state p is pushed into the incoming transitions of p. We can accumulate all the weights pushed through each state p to be the potential weight on p. This is, of course, still very inefficient due to algorithm WEIGHT-REDISTRIBUTION-BY-PUSHING.

Here we propose in figure 3.5 an efficient algorithm which computes the potential weights directly rather than computing by the weight-pushing based algorithm.

In algorithm GENERALIZED-FLOYD-WARSHALL, D_{ij}^{k} means the greatest com-

Algorithm WEIGHTS-REDISTRIBUTION-EFFICIENTLY

Require: A be the input automaton

Ensure: B be the automaton after weight redistribution

1: $B \leftarrow A$

2: **for all** $p \in S$ **do**

3: **for all** $q \in S$ **do**

4: $W[p, q] \leftarrow \infty$

5: **for all** $t \in T$ **do**

6: $W[p, q] \leftarrow gcd\_of(W[p, q], weight(t))$

7: **end for**

8: **end for**

9: **end for**

10: $d \leftarrow compute\_potential\_function(W)$

11: **for all** weight w of transitions t in B **do**

12: $w \leftarrow (w \otimes d(dst(t), f)) \oslash d(src(t), f)$

13: **end for**

14: **return** B

Figure 3.4: A weight redistributing algorithm by setting potentials.

Algorithm Generalized-Floyd-Warshall

Require: W be the input transition matrix, F be the final weights.

Ensure: d be the generalized shortest distance function

1: $m \leftarrow$ # rows in W

2: **for** $i = 1$ to m **do**

3: $\quad W[i, m+1] \leftarrow F[i]$

4: $\quad W[m+1, i] \leftarrow \bar{1}$

5: **end for**

6: $n \leftarrow m + 1$

7: $D^0 \leftarrow W$

8: **for** $k \leftarrow 1$ to m **do**

9: $\quad$ **for** $i \leftarrow 1$ to n **do**

10: $\quad\quad$ **for** $j \leftarrow 1$ to n **do**

11: $\quad\quad\quad D_{ij}^k \leftarrow gcd\_of(D_{ij}^{k-1}, D_{ik}^{k-1} \otimes D_{kj}^{k-1})$

12: $\quad\quad$ **end for**

13: $\quad$ **end for**

14: **end for**

15: Let $d(i)$ denotes D_{in}^n

16: **return** d

Figure 3.5: Compute the potential function by generalizing the Floyd-Warshall algorithm.

mon divisor among pathes from state i to state j for which all intermediate states are in the set $\{1, 2, ..., k\}$. When $k = 0$, a path from i to j with no intermediate state numbered higher than 0 has no intermediate states at all. Such a path has at most one transition, and hence $D^0_{ij} = w(i \to j)$. Combined with the above discussion, we have D^k_{ij} recursively:

$$D^k_{ij} = \begin{cases} w(i \to j), & \text{if } k = 0 \\ gcd\_of(D^{k-1}_{ij}, D^{k-1}_{ik} \otimes D^{k-1}_{kj}), & \text{if } k > 0 \end{cases}$$

Because for any path, all intermediate states are in the set $\{1, 2, ..., n\}$, the matrix D^n gives the final answer: $D^n_{in} = d(i), \forall i \in S$. So we are able to obtain the following lemma.

Lemma 1. *In an weighted automaton* $M = (K, Q, T, \Sigma, I, F), \forall p \in Q$, *algorithm* Generalized-Floyd-Warshall *computes* $d(.)$ *such that*

$$d(p) = gcd\_of(w(y) : y \text{ is a successful path starting from } p.)$$

Or equivalently,

$$d(p) = gcd\_of(d(dst(t)) \otimes w(t)), \forall t \in out(p)$$

From another point of view, this algorithm works because, like $min(.)$ defined on $+$ in the classical Floyd-Warshall algorithm, $gcd\_of(.)$ defined on $\otimes$ is also an order which is required by the recursive definition of D^k. In other words, they all computes the common infimum, namely, the greatest common lower bound. The only difference between classical Floyd-Warshall algorithm and algorithm Generalized-Floyd-Warshall is, in the classical Floyd-Warshall algorithm:

$$D^k_{ij} \leftarrow min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj})$$

and, in algorithm Generalized-Floyd-Warshall:

$$D_{ij}^{k} \leftarrow gcd\_of(D_{ij}^{k-1}, D_{ik}^{k-1} \otimes D_{kj}^{k-1})$$

It is easy to see that the $min()$ is replaced by $gcd\_of()$ and $+$ is replaced by $\otimes$. In fact, algorithm Generalized-Floyd-Warshall generalize the classical Floyd-Warshall algorithm by solving the generalized shortest path problem. If we specialize algorithm Generalized-Floyd-Warshall on the semiring $(N, +, min, 0, \infty)$, it is exactly the same with the classical Floyd-Warshall algorithm where gcd_of degenerates to min.

In $(N, *, +, 1, 0)$, for example, we can express each number by products of powers of primes in vectors. For instance, $84 = 2^2 * 3^1 * 5^0 * 7^1$ and vector $v_{84} =< 2, 1, 0, 1 >$. By define $min()$ on two vector v and u by

$$min(v, u) =< min(v_1, u_1), ..., min(v_i, u_i), ..., min(v_n, u_n) > .$$

we can write g.c.d. by $gcd\_of(m, n) = min(v_m, v_n)$.

Although in the beginning our goal is to find efficient alternitives to the weight-pushing based algorithm, this algorithm does do more than the weight-pushing based algorithm. It compute the greatest common divisors of the weights associated by all possible successful pathes rather than just outgoing transitions as in the later case.

Theorem 2. *On weighted automata whose weights are semirings on which greatest common divisors are properly defined, algorithm* Weights-Redistribution-Efficiently *which uses algorithm* Generalized-Floyd-Warshall *redistributes weights such that the greatest common divisors of weights associated by all possible successful path starting from each states are* $\bar{1}$.

Also that algorithm WEIGHTS-REDISTRIBUTION-EFFICIENTLY *don't violate the closure of the semirings and is language preserving.*

Proof. We prove the second part first. Assume we are given transition t. Because $d(src(t)) = gcd\_of(d(dst(t)) \otimes w(t), ...)$, $d(src(t))$ div $(d(dst(t)) \otimes w(t))$. So we have $(w(t) \otimes d(dst(t))) \oslash d(src(t))$ must be closed in the same semiring.

For weight $w_{old}(y)$ and $w_{new}(y)$ before and after applying the algorithm respectively of any successful path $y = p_1 \to p_2 \to ... \to p_n$, $w_{old}(y) = initial(p_1) \otimes w(p_1 \to p_2) \otimes ... \otimes final(p_n) = initial(p_1) \otimes d(p_1) \otimes w(p_1 \to p_2) \oslash d(p_1) \otimes ... \otimes final(p_n) \oslash d(p_n) = w_{new}(y)$.

We prove the first part by contradictory. Assume there exists a state p such that $gcd\_of(w(y) : y \in Succ(p)) = c > \bar{1}$ after applying algorithm WEIGHTS-REDISTRIBUTION-EFFICIENTLY which uses GENERALIZED-FLOYD-WARSHALL. Then before changing the weights of transitions it must be the case that $gcd\_of(w(y) : y \in Succ(p)) = c \otimes d(p)$ because we divide each outgoing transitions of p by $d(p)$ in the algorithm. This is a contradiction to $d(p)$ by lemma 1. □

What the two algorithm differing is that, in a few words, algorithm WEIGHTS-REDISTRIBUTION-BY-PUSHING pushes the greatest common divisors of weights of all outgoing transitions of each states. Algorithm WEIGHTS-REDISTRIBUTION-EFFICIENTLY which uses algorithm GENERALIZED-FLOYD-WARSHALL pushes the greatest common divisors of weights associated by all successful pathes starting from each states. Obviously algorithm WEIGHTS-REDISTRIBUTION-BY-PUSHING is a special case and less powerfull than algorithm WEIGHTS-REDISTRIBUTION-EFFICIENTLY which is equiped with algorithm GENERALIZED-FLOYD-WARSHALL.

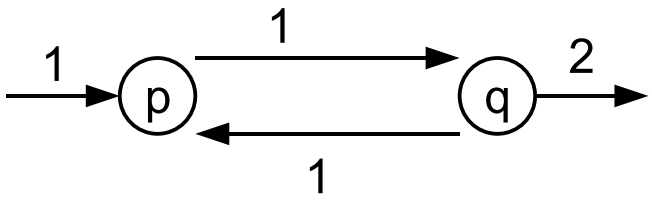


Figure 3.6: An exmple resulted differently by these two algorithms.

Hence we have the following corollary.

Corollary 2. *If there is a state whose transitions are with greatest common divisor greater than* 1*, it must be discoverd by algorithm* Weights-Redistribution-Efficiently *which uses algorithm* Generalized-Floyd-Warshall.

That is, after applying algorithm Weights-Redistribution-Efficiently which uses algorithm Generalized-Floyd-Warshall, there remains no states on which we can push weights.

To show algorithm Generalized-Floyd-Warshall performs better, we give an example in figure 3.6. Obviously it is impossible to push q's final weights which is the only weight greater than 1. The weight-pushing based algorithm doesn't help any further. In algorithm Generalized-Floyd-Warshall, both p and q's potential function will be set to 2 so eventually the 2 in the final weight of q will be redistributed to the initial weight of p.

Chapter 4

The Minimizing Algorithm

The most famous study of determinimizing the equivalence of and minimizing deterministic finite automata is perhaps the Myhill-Nerode theorem. The Myhill-Nerode theorem also helps on reducing the state space of nondeterministic finite automata by the notion of Nerode quotient. The minimization, not necessary to the minimal form on nondeterministic ones, relies mainly on the local structures in the targetted automata. To be more precise, it requires some properties of blocks of states so that we can merge all states in a block while preserving the same language.

For example, we can merge state p and state q in figure 4.1 because their "future" are the same. That is, for every remaining input string w while current state is at p and q respectively, they always lead to the same results which is either accept or reject. Along this way, we can't distinguish p and q in terms of the strings this automaton accepts, i.e., in terms of language this automaton defined so we are able to merge p and q by adding up their incoming transitions.

One of the important results on deterministic finite automata is that, for two

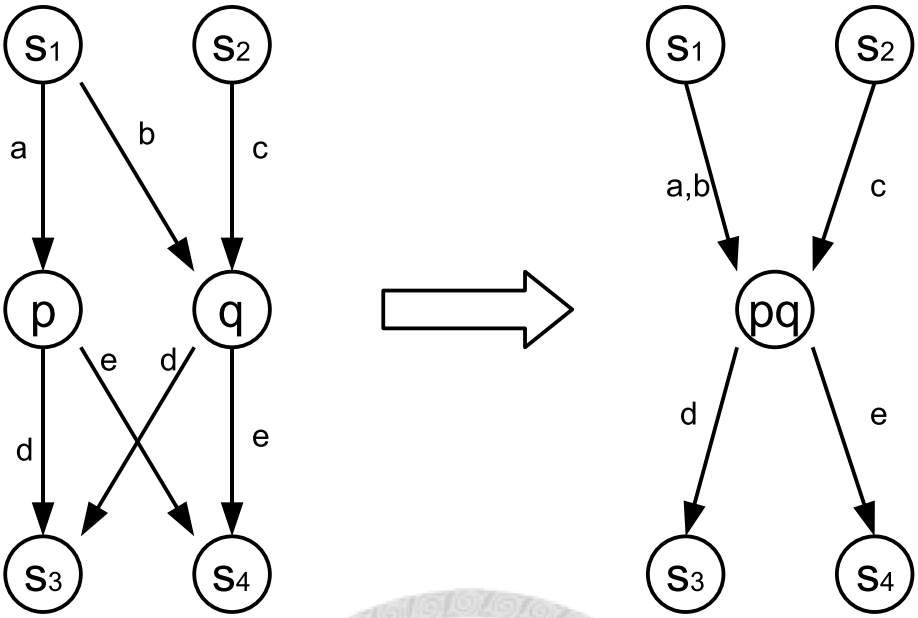


Figure 4.1: The Nerode quotient on DFA.

states p and q to have the same future, their outgoing transitions must be the same in terms of the quotient. For example, if p has an outgoing transition to r in block g, q must have an outgoing transition to s in g. This property is usually refered to as bisimulation relation. Most of the algorithms is based on this property [14, 19].

But things are complicated when we are considering the nondeterministic cases. The minimal form of a nondeterministic finite automaton can no longer be computed by bisimulation relation though that it sometimes helps on reducing the state space. In fact, for p and q to have the same future, it is no longer required that they are bisimulation equivalent. Also that there is no promise that the minimal forms can be found by merging states that are Nerode equivalent. Even that computing Nerode equivalence becomes harder, too.

We briefly compare the power and complexity of the methods on nondeterministic

Table 4.1: Reducing state space of NFA

Goal	Power	Comlexity
bisimulation equivalence	less powerful	$O(N \lg N)$
Nerode equivalence	more powerful	P-Space complete
Language equivalence	ultimate goal	P-Space complete

automata mentioned in the above in table 4.1.

Like that in table 4.1, we are going to augment the methods of minimizing nondeterministic weighted finite automata from bisimulation equivalence toward Nerode quotient. In other words, our goal is to check if two states that are mergiable as possible.

In this chapter, we firstly introduce the algorithm proposed in [12] and [20] which are based on the notion of Nerode quotient. We then show that it is possible to reduce the state space further by making use of the local structure more of the targetted automata in section 4.2. In section 4.3 we combine the efforts, that is, the algorithm proposed in the previous chapter to give a heuristic algorithm which perform much better than that in [20] and [12]. In section 4.4 we show that computing quotient and co-quotient are equivalent. That is, we can compute co-quotient by reducing it to a quotient problem. Also that our algorithms developed in the previous chapter can be applied in the other direction as well.

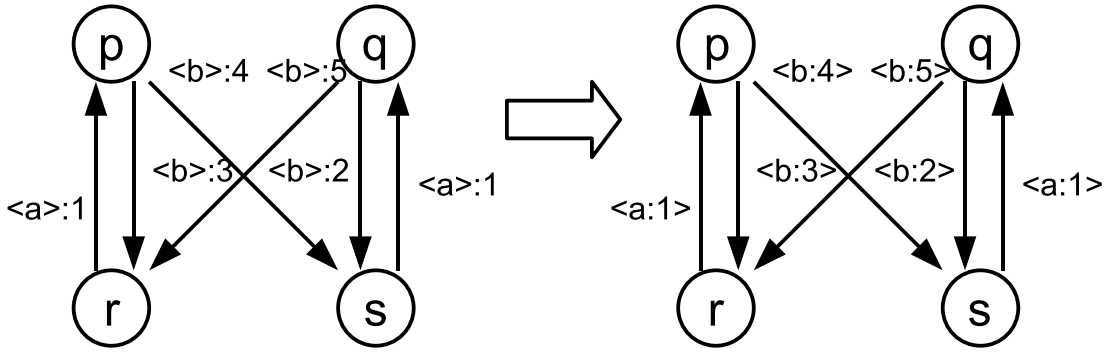


Figure 4.2: Mohri's method to compute the Nerode quotient

4.1 Computing Nerode Quotient

On defining bisimulation relations of weighted finite automata, the most intuitively way is to treat the weights as "independent symbols". As in [12], this method forms new labels from $(original\ label, weight)$ pairs to replace all the original labels in weighted finite automata before minimizing the classical finite automata.

In figure 4.2, for example, if there exist transition $t = (p, q, l, w)$, which is a transition from p to q with label a and weight w, then $t = (p, q, l, w)$ is transformed to $(p, q, (l, w))$ which is a transition of some standard finite automaton so that we can apply minimization techniques on standard finite automata such as [12].

Although this method is very easy to understand and to implement and as fast as that on standard finite automata, its perfromance is not as good as its simplicity. One of the most important drawback is that we can't take any advantage of the fact that weights in a weighted finite automaton are actually in a semiring. This simple method breaks the algebraic connections among the weights in weighted finite

Algorithm K-QUOTIENT

Require: $A = (K, Q, \Sigma, T, I, F)$ be the input automaton.

1: Partition Q into P_i by their final weights.

2: **repeat**

3: **if** $\exists p, q \in P_i$ such that $\sum_{t \in out-to-X(p)} weight(t) \neq \sum_{s \in out-to-X(q)} weight(s)$ for some other partition X **then**

4: split P_i

5: **end if**

6: **until** the structure becomes stable

Figure 4.3: Computing K-Quotient.

automata and make very poor use of the algebraic structures.

The algorithm in [20] is in fact a generalization to weighted finite automata of Hopcroft's $O(N \lg N)$ algorithm [8]. It computes K-quotient defined on weighted finite automata in [22]. As most minimization algorithms to DFA, it makes use of the bisimulation relation which is an equivalence relation to derive the partition of the states then to merge states in the same blocks. We give the central ideas by showing the pseudo codes in figure 4.3. This algorithm is almost the same with Hopcroft's algorithm except two of the following.

First, it sets up the initial partition by each state's final weights, rather than the acceptance in Hopcroft's algorithm. If state p's and q's final weight's are different, they are put in different blocks. Second, it splits the blocks, or equivalently, decides whether states are the same, by calculating the total weights of all transitions for

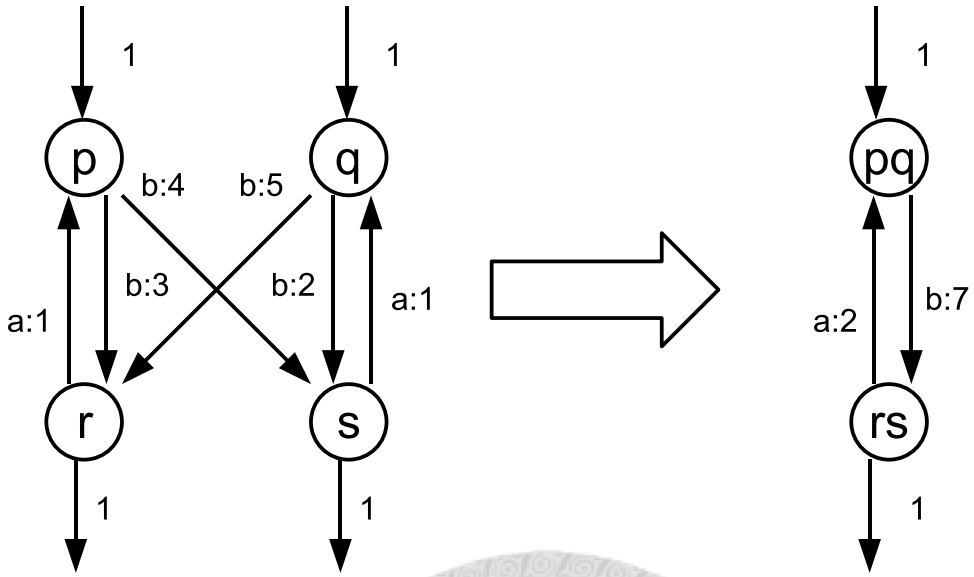


Figure 4.4: Merge p and q, r and s by K-quotient

all labels and blocks. This is the natural restriction that causes this algorithm runs in $O(N^2)$ rather than $O(N \lg N)$ as Hopcroft's algorithm.

For example, state p and q in figure 4.4 can be merged in this algorithm. Initially the states $\{p, q, r, s\}$ is partitioned into $\{\{p, q\}, \{r, s\}\}$ by their final weights. So p and q are pontentially equivalent as well as r and s. Since $\forall a \in \Sigma^*, \forall t \in \{out(p) : label(t) = a\}, \forall u \in \{out(q) : label(u) = a\}, \sum_t weight(t) = 7 = \sum_u weight(u)$, we can't split $\{p, q\}$. Similary we can't split $\{r, s\}$. By theorem 3.7 in [15], the converged partition $\{\{p, q\}, \{r, s\}\}$ is the Nerode equivalence relation. So we are able to merge p and q as well as r and s by adding up their incoming transitions.

Obviously this algorithm computing K-quotient is strictly powerful then that in [12]. Also that it is well implemented in the Vaucanson project [20] such that it is applicable to weighted automata with weights over dozens of types of semirings.

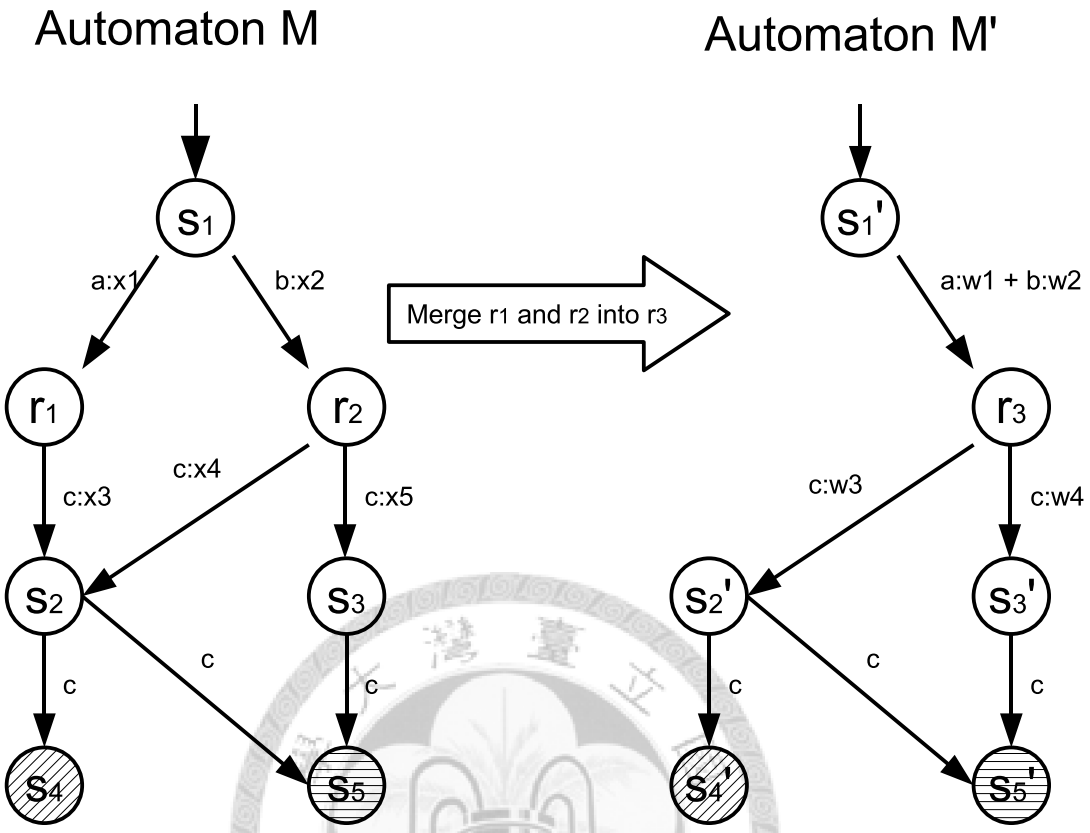


Figure 4.5: Requirements to merge states

4.2 The Parallel Structure

In general, if two states in an weighted finite automaton are to be merged, they must satisfy some constraints. Before stating the constraints let's first see an example to get some intuition.

In figure 4.5, r_1 and r_2 are merged into r_3. The weights are in $(N, *, +, 1, 0)$. Let (p, q, l, w) denotes the transition from p to q with label l of weight w.

After merging r_1 and r_2 into r_3, let $(s_1', r_3, a, w1)$, $(s_1', r_3, b, w2)$, $(r_3, s_2', c, w3)$, $(r_3, s_3', c, w4)$ be the new transitions in M'.

For M', we have $M'(\text{“}ac\text{”}) = w_1 * (w_3 + w_4)$, $M'(\text{“}bc\text{”}) = w_2 * (w_3 + w_4)$, $M'(\text{“}acc\text{”}) = w_1 * (2 * w_3 + w_4)$, $M'(\text{“}bcc\text{”}) = w_2 * (2 * w_3 + w_4)$. Obviously,

$$\frac{M'(\text{“}ac\text{”})}{M'(\text{“}bc\text{”})} = \frac{w_1}{w_2} = \frac{M'(\text{“}acc\text{”})}{M'(\text{“}bcc\text{”})}.$$

For M, we have $M(\text{“}ac\text{”}) = x_1 * x_3$, $M(\text{“}bc\text{”}) = x_2 * (x_4 + x_5)$, $M(\text{“}acc\text{”}) = 2 * x_1 * x_3$, $M(\text{“}bcc\text{”}) = x_2 * (2 * x_4 + x_5)$. To realize the same series, we must have

$$\frac{M(\text{“}ac\text{”})}{M(\text{“}bc\text{”})} = \frac{x_1 * x_3}{x_2 * (x_4 + x_5)} = \frac{2 * x_1 * x_3}{x_2 * (2 * x_4 + x_5)} = \frac{M(\text{“}acc\text{”})}{M(\text{“}bcc\text{”})}.$$

That is,

$$x_5 = 0.$$

There are indeed some constraints between r_1 and r_2 to be merged.

As suggested in figure 4.5, r_1 and r_2 must have their “future” be compatible to each other. Here we consider “future” to be following.

Definition 12. *$\forall w \in \Sigma^*$, M_p denotes running M with only starting states p and initial weight* 1.

For example, in figure 4.5 $M_{r_2}(\text{“}c\text{”}) = 1 * x4 * final(s_2) + 1 * x5 * final(s_3)$.

Now we can describe formally what is needed to merge states when the operations allowed are union and reweighting both their incoming and outgoing transitions, if we only consider the constraints on their outgoing transitions.

Corollary 3. *For any two states p and q to be merged by union and reweighting their incoming and outgoing transitions, it must be the case that*

$$\forall w \in \Sigma^*, M_p(\text{“}w\text{”}) * c_p = M_q(\text{“}w\text{”}) * c_q$$

for some fixed c_p and c_q, if we only consider the requirements on p and q's outgoing transitions. We say that p and q satisfying the above equation are compatible to each other.

It seems to be a good property that we may start to design an algorithm. Unfortunately, we can easily reduce the problem of determining the equivalence of two weight finite automata M and N to the problem of determining whether this property holds between state p and q by adding p and q to be the only initial states of M and N respectively: if p and q are proportational to each other and $c_p = c_q$, then $M = N$. So deciding whether the property hold between two states is hard.

We have to specialize, or namely, tighten up the constraint. This constraint, which we called parallel structure, is the similarity of outgoing transitions of each states.

Definition 13. *Let $W(s,t,l)$ be the weight of state p's outgoing transition (s,t,l,w), namely $W(s,t,l) = w$ in (s,t,l,w). State p and state q are said to be parallel to each other if and only if $\exists c_p, \exists c_q, \forall r, \forall l, c_p * W(p,r,l) = c_q * W(q,r,l)$*

In other words, two states are parallel to each other if their outgoing transitions are proportional to each other with a common factor. In figure 4.6, state p and q are parallel to each other.

Now the constraint is put on the outgoing edges rather than "futures". Since the parallelism between p and q implies that the set of their successors are the same, it is easy to see that the parallelism of p and q also implies the compatibility between p and q.

Corollary 4. *If p and q are parallel to each other, then p and q are compatible.*

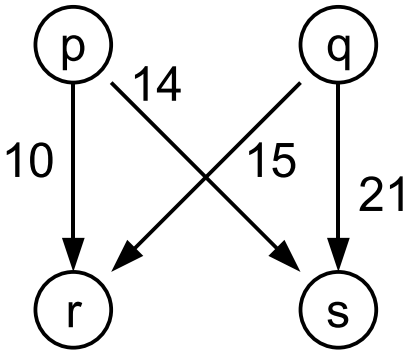


Figure 4.6: The parallel structure

In figure 4.7, we show that it is always possible to merge two states which are parallel to each other. By pushing the weight of $p \to s_3$ and $p \to s_4$ to $s_1 \to p$ and $s_2 \to p$ by factor c, $q \to s_3$ and $q \to s_4$ to $s_1 \to q$ and $s_2 \to q$ by factor d, we successfully reduce the slightly complicated case to the trivial case on which two states' outgoing transitions are exactly the same. Now we can easily merge them using algorithms proposed in [12] and [20].

4.3 Our Algorithm

Now we are ready to give the algorithm to minimize weighted automata.

This algorithm simply contains two steps. It redistribte weights of the input automata then compute the K-quotient. We show that this algorithm discovers all the parallel structures and transform them into the trivial cases in the first step and make use of the results by first step to shrink the state space in the second step.

Theorem 3. *Algorithm* Weights-Redistribution-Efficiently *which uses* Generalized-Floyd-Warshall *discovers all the parallel structures and transforms them into the*

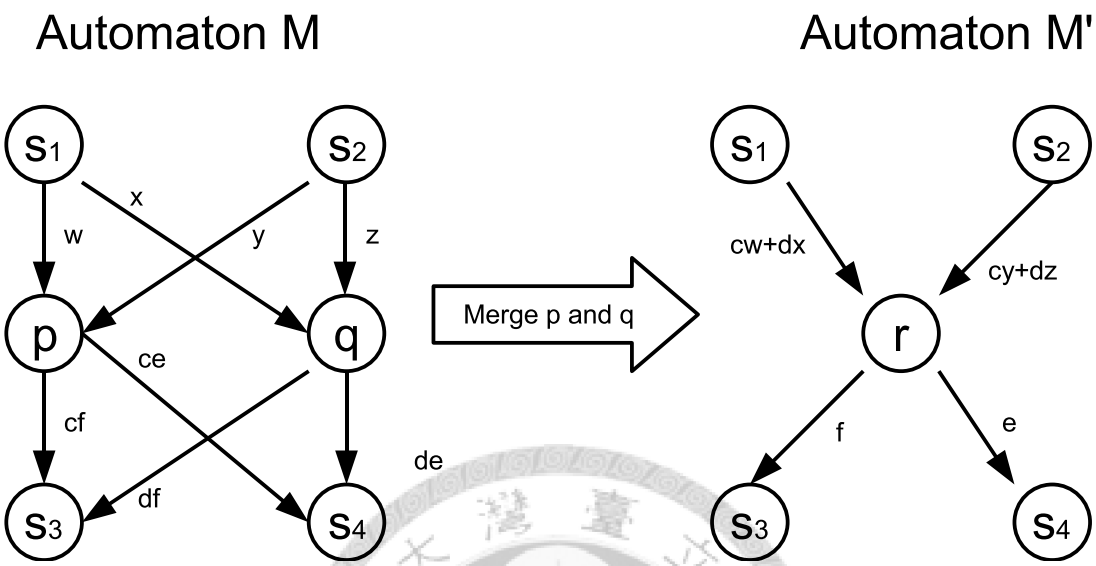


Figure 4.7: Minimization utilizing the parallel structure

Algorithm MINIMIZE-WFA

Require: A be the input automaton.

1: Redistribute weights of A

2: Compute K-quotient of A

Figure 4.8: The minimizing algorithm.

trivially mergeable cases while the greatest common divisor are definable.

Proof. If we see the outgoing transitions of states p and q as vectors denoted by $v(p)$ and $v(q)$ respectively, there exist c_p and c_q such that $c_p \otimes v(p) = c_q \otimes v(q)$ by definition. There must exist some vector v_s such that $v_s \otimes lcm(c_p, c_q) \otimes c_s = c_p \otimes v(p) = c_q \otimes v(q)$ where $lcm()$ means least common multiplier and the greatest common divisor of the elements of v_s is 1. When pushing weights implicitly by algorithm GENERALIZED-FLOYD-WARSHALL on state p and q, the greatest common factors of each set of outgoing transitions are $lcm(c_p, c_q) \oslash c_q \otimes c_s$ and $lcm(c_p, c_q) \oslash c_p \otimes c_s$ respectively. After weight pushing, the outgoing transitions of both states becomes v_s.

So, if there are two states which are parallel to each other, they must be discovered and transformed to the trivially mergeable cases by algorithm GENERALIZED-FLOYD-WARSHALL. □

The first part of this algorithm runs in $O(N^3)$ if we can find the greatest common divisor of two elements in the semiring in constant time. The second part runs in $O(N^2)$ as explained in section 4.1 and in [20]. So this algorithm runs in $O(N^3)$.

4.4 The Backward Counterparts

Until this section, all we talk about are constraints on states' outgoing transitions. We now show that it is also benefited by looking in the reverse direction. That is, to consider about incoming transitions rather than outgoing transitions.

From now on we refer to algorithms that utilize special conditions of local structures of outgoing transitions as **forward counterparts** and that utilize incoming

transitions as **backward counterparts**.

First of all, the operations and algorithms developed in chapter 3 are all symmetric. Although they might have different meanings, they do be language preserving when applied in the reverse direction.

Now let's see the bisimulation relation on standard automata in the reverse direction. This is usually called backward-bisimulation [15]. It is, as the readers might expect, language preserving, too. It is obvious that if state p and q have the same "history", that is, on executing an automaton, if the current state is at p it must be also at q then they can be merged by adding up their outgoing transitions to obtain the same language.

For weighted finite automata, we give an example to illustrate that computing K-quotient reversely, or namely, computing co-K-quotient, benefits on reducing the state space and is language preserving.

In figure 4.9, state p and q, r and s can be merged in since their co-K-quotient are the same respectively. Initially the states $\{p, q, r, s\}$ is partitioned into $\{\{p, q\}, \{r, s\}\}$ by their initial weights. So p and q are pontentially equivalent as well as r and s. Since $\forall a \in \Sigma^*, \forall t \in \{in(p) : label(t) = a\}, \forall u \in \{in(q) : label(u) = a\}, \sum_t weight(t) = 7 = \sum_u weight(u)$, we can't split $\{p, q\}$. Similary we can't split $\{r, s\}$. By theorem 3.7 in [15], the converged partition $\{\{p, q\}, \{r, s\}\}$ is the Nerode equivalence relation. So we are able to merge p and q as well as r and s by adding up their outgoing transitions.

Quite satisfying results in the above while we still have some remarks about computing co-K-quotients. One is that computing co-K-quotients, or equivalently

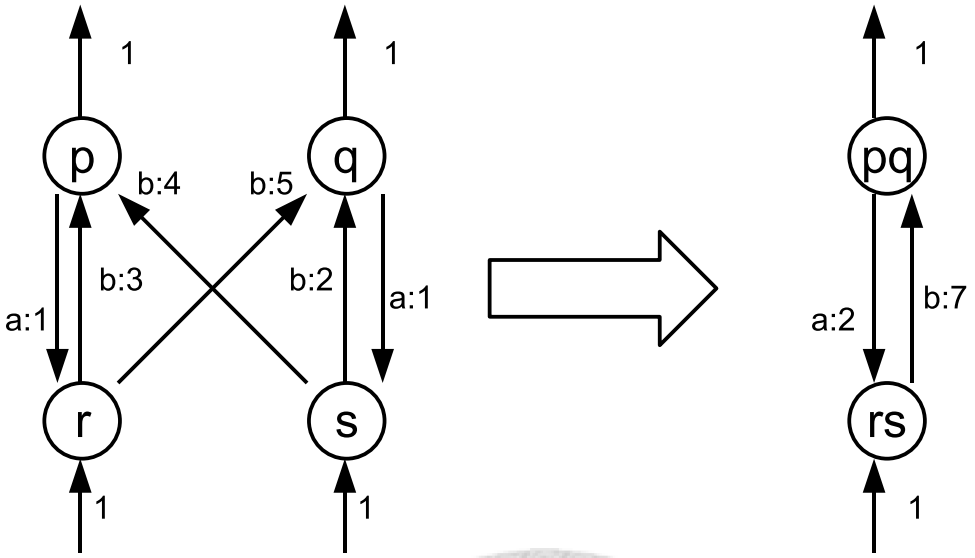


Figure 4.9: Minimization with co-K-quotient

backward bisimulation equivalence, is totally useless on standard deterministic finite automata as well as deterministic weighted finite automata. Because in a deterministic one, states never share common predecessors nor histories, otherwise it would violate the definition of determinism. The other is that on merging states by these backward counterparts, they increse the nondeterminism by adding up the outgoing transitions. It's also easy to see that merging by those forward counterparts don't add any nondeterminism to automata.

To make connection between the forward and backward counterparts, we define an operation Transpose on weighted finite automata in figure 4.10. In a few words, what Transpose does is that, the direction of transitions are reversed and initial weights become final weights, final weights become initial weights.

To end this section, we give one proposition without proof. we denote forward-

Algorithm TRANSPOSE

Require: A be the input automaton.

Ensure: $A \leftarrow transpose(A)$

1: **for all** transition t in A **do**

2: $s \leftarrow dst(t)$

3: $dst(t) \leftarrow src(t)$

4: $src(t) \leftarrow s$

5: **end for**

6: **for all** state p in A **do**

7: $s \leftarrow initial(p)$

8: $initial(p) \leftarrow final(p)$

9: $final(p) \leftarrow s$

10: **end for**

11: **return** A

Figure 4.10: Transpose().

bisimulation-minimization by $fbm(.)$, backward-bisimulation-minimization by $bbm(.)$, forward-weight-redistribution by $fwr(.)$, backward-weight-redistribution by $bwr(.)$. One may check the pseudo codes of the algorithms mentioned in the following to see the correctness of this proposition easily.

Proposition 3. *The connections between forward and backward counterparts can be described by*

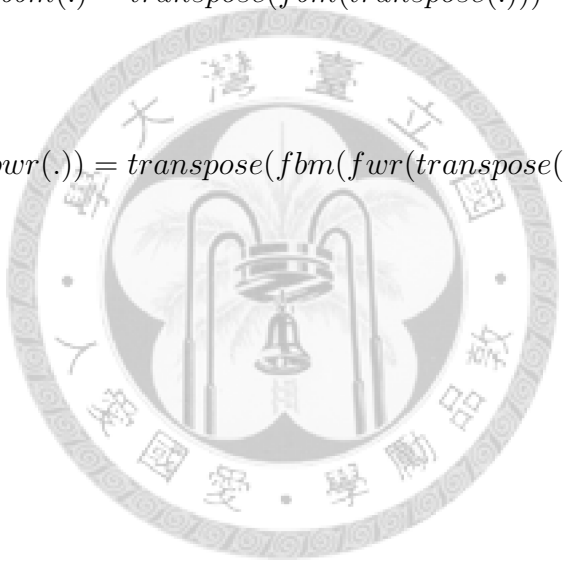
$$bwr(.) = transpose(fwr(transpose(.))),$$

$$bbm(.) = transpose(fbm(transpose(.)))$$

and

$$bbm(bwr(.)) = transpose(fbm(fwr(transpose(.))))$$

and vice versa.



Chapter 5

Experimental Results

This experiment is carried out on the Vaucanson library version 1.2 which is designed to be very general in terms of semirings of weights of weighted finite automata [21, 23].

Vaucanson is a finite automata, or equivalently finite state machines, manipulation platform which is initiated by Jacques Sakarovitch and Sylvain Lombardy in 2001. In earlier days, such platforms were developed to work either at some industrial scale, designated to weighted letter automaton to be efficient [24] or in a pure abstract way [25]. Via static and generic C++ programming, Vaucanson tries to respond to these two trends.

Vaucanson is targeting at automata with multiplicity over any semiring. A general algorithm can be written just once and can be statically instantiated for any particular kind of automaton. As a result, efficient codes are obtained from algorithms written in such an abstract way using basic primitives taken from the C++ language.

More concretely, Vaucanson copes with generic algebraic structures including polynomials as maps from monoid values to weights and rational expressions as binary trees. It deals with automaton structures by using graph structures that ensures minimum complexity for most operations. There are also many common and well known algorithms on automata, for example, determinization, quotient, product, ϵ-removal and many more, implemented in Vaucanson. There are also generic testsuites for user-made extensions in Vaucanson.

We have done this experiment in Linux 2.6.22 on an Intel Core 2 Duo 1.83GHz machine with 3GB system memory. The compiler we used is gcc-4.2.3 with -O2 flag. The speed, or efficiency, is quite satisfying that all the test cases are finished within 0.1 seconds except for the weight-pushing based weight redistributing algorithm which cost less than 1 seconds among several runs of the worst cases in this experiment.

We compared three of the algorithms which are K-quotient, Minimize-WFA with Weights-Redistribution-By-Pushing and Minimize-WFA with Weights-Redistribution-Efficiently with Generalized-Floyd-Warshall. In the following figures, they are abbreviated as KQ, BF and FW respectively. Also that we compared the performance of applying these algorithms forward, backward and both. So there are totally 9 combinations that are, F-KQ, B-KQ, FB-KQ, F-BF, B-BF, FB-BF, F-FW, B-FW and FB-FW.

Our experiment targets at weighted finite automata over $N = (N, +, *, 0, 1)$. We follow the method by Zijl in [27, 28] to generate nondeterministic finite automata with a little modification and randomly add weights in N as following:

- The alphabet is $\Sigma = 1, 2, ..., m$, the set of states is $Q = 1, 2, ..., n$ and the range of possible weight values in the beginning is $W = [1, o]$.

- An equiprobable bitstream of size mn^2 is generated. It descirbes the transition function δ that the occurrence of a non-zero bit at position $(l-1)n^2+(i-1)n+j$ denotes the existence of a transition from state i to state j and labeled by l.

- Randomly assign $w \in W$ to each transition, initial and final weights of each state.

The difference between Zijl's method and ours is that we randomly assign weights to initial and final weights of each state rather than randomly deciding initial and final states in Zijl's method. Also that we have to add weights to each transitions, randomly.

The density, definied in [27, 28], of a nondeterministic weighted finite automaton with e transitions is $da = \frac{e}{mn^2}$. Here we adopt another definition of density $d = \frac{e}{n}$ because in the following experiments it is shown that the compressibility of weighted finite automata is closely related to $\frac{e}{n}$ and to be independent of m.

The test cases are generated by 4 of different transition densities, $d = 0.5, 1.0, 1.5, 2.0$, 4 of ranges of possible weight values in the beginning, $W = [1, 2], [1, 4], [1, 8], [1, 16]$ and 4 of alphabet sizes, $\Sigma = \{1\}, \{1, 2\}, \{1, 2, 3, 4\}, \{1, ..., 8\}$. Without loss of generality, we fix the number of states to be 100.

We also test the performance over 3 common operations: concatenation, union and product on weighted finite automata. We concatenate two different weighted finite automata which have 100 states with $d = 1.0, m = 2, w = 8$ so it has 200 states and 200 transitions before minimization. We union two different weighted

finite automata which have 100 states with $d = 1.0, m = 2, w = 8$. We product two different weighted finite automata which have 20 states with $d = 1.0, m = 2, w = 8$.

The experiments reveal something:

- Our algorithms have apparently better perfromance than K-quotient in every sets of parametersa. This is mainly due to the stronger condition required by our algorithms on the underlying semirings and our algorithms successfully make use of these conditions.

- The readers may note that in K-quotient, the compressibility of automata is hightly related to the ranges of weights in the beginning. Our algorithms ease this problem, that is, the compressibility of automata becomes slightly related or even unrelated to the ranges of weights in the beginning. This is because our algorithms can make use of the connections between elements in semirings. In other words, since the automata in the experiments are relatively sparse (for $d = \Omega(n)$ it becomes uncompressible), there is high probability that weights are redistributed to initial or final and the resulting structures are simple enough for states to merge.

- Both in K-quotient and our 2 algorithms, the compressibility seems very insensitive to the size of alphabets. This may due to that our density is fixed to number of transitions. The complexity of graph structures mainly depends on total number of transitions and slightly on number of kinds of transitions.

- Our algorithms are more effective with concatenation, union and product, especially with product.

Table 5.1: Experimental results with $m = 1, w = 2$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	29	27	18	**24**	**21**	**15**	**23**	**21**	**15**
1.0	60	54	41	**55**	**48**	**34**	**51**	**47**	**33**
1.5	72	78	63	**68**	**76**	**60**	**64**	**71**	**56**
2.0	88	88	79	**87**	**87**	**77**	**86**	**84**	**74**

Table 5.2: Experimental results with $m = 1, w = 4$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	40	41	26	**32**	**32**	**18**	**31**	**30**	**17**
1.0	67	64	49	**61**	**55**	**39**	**58**	**52**	**37**
1.5	78	83	68	**73**	**79**	**61**	**68**	**73**	**58**
2.0	91	90	83	**88**	**87**	**78**	**85**	**84**	**75**

Table 5.3: Experimental results with $m = 1, w = 8$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	46	50	33	**35**	**39**	**20**	**33**	**36**	**19**
1.0	73	68	56	**65**	**59**	**42**	**62**	**56**	**40**
1.5	82	85	74	**75**	**80**	**62**	**72**	**76**	**59**
2.0	93	92	87	**88**	**87**	**78**	**82**	**82**	**74**

Table 5.4: Experimental results with $m = 1, w = 16$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	54	58	45	**38**	**43**	**20**	**35**	**41**	**19**
1.0	80	74	66	**66**	**62**	**43**	**65**	**60**	**42**
1.5	88	90	82	**75**	**80**	**62**	**74**	**79**	**60**
2.0	97	96	94	**88**	**87**	**78**	**84**	**83**	**76**

Table 5.5: Experimental results with $m = 2, w = 2$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	33	31	19	**30**	**27**	**16**	**30**	**27**	**16**
1.0	62	58	44	**57**	**55**	**38**	**56**	**54**	**38**
1.5	74	79	64	**70**	**78**	**61**	**66**	**74**	**57**
2.0	89	88	80	**88**	**87**	**78**	**84**	**83**	**74**

Table 5.6: Experimental results with $m = 2, w = 4$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	45	41	24	**33**	**33**	**15**	**32**	**32**	**15**
1.0	68	64	50	**62**	**57**	**41**	**59**	**54**	**39**
1.5	78	83	68	**73**	**79**	**61**	**70**	**74**	**57**
2.0	91	90	83	**88**	**87**	**78**	**84**	**83**	**73**

Table 5.7: Experimental results with $m = 2, w = 8$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	49	45	32	**41**	**36**	**17**	**40**	**35**	**16**
1.0	73	68	56	**65**	**60**	**43**	**63**	**58**	**42**
1.5	82	85	74	**75**	**80**	**62**	**72**	**77**	**59**
2.0	93	92	87	**88**	**87**	**78**	**84**	**83**	**75**

Table 5.8: Experimental results with $m = 2, w = 16$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	57	52	44	**41**	**38**	**18**	**40**	**37**	**18**
1.0	80	74	66	**66**	**62**	**43**	**62**	**58**	**41**
1.5	88	90	82	**75**	**80**	**62**	**73**	**78**	**61**
2.0	97	96	94	**88**	**87**	**78**	**86**	**85**	**76**

Table 5.9: Experimental results with $m = 4, w = 2$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	36	37	19	**31**	**33**	**15**	**29**	**31**	**14**
1.0	65	61	45	**60**	**57**	**40**	**57**	**54**	**38**
1.5	75	80	64	**72**	**79**	**61**	**68**	**75**	**58**
2.0	89	88	80	**88**	**87**	**78**	**83**	**82**	**73**

Table 5.10: Experimental results with $m = 4, w = 4$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	45	41	24	**37**	**38**	**17**	**35**	**36**	**16**
1.0	69	65	49	**64**	**60**	**42**	**62**	**58**	**41**
1.5	85	78	69	**82**	**75**	**63**	**81**	**74**	**62**
2.0	91	90	83	**88**	**87**	**78**	**85**	**81**	**72**

Table 5.11: Experimental results with $m = 4, w = 8$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	49	45	32	**42**	**38**	**18**	**40**	**36**	**17**
1.0	73	68	56	**66**	**60**	**43**	**65**	**59**	**42**
1.5	89	82	76	**82**	**75**	**63**	**80**	**73**	**62**
2.0	93	92	87	**88**	**87**	**78**	**84**	**86**	**77**

Table 5.12: Experimental results with $m = 4, w = 16$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	57	52	44	**42**	**38**	**18**	**40**	**36**	**17**
1.0	80	74	66	**66**	**62**	**43**	**65**	**61**	**43**
1.5	94	87	84	**82**	**75**	**63**	**81**	**72**	**61**
2.0	97	96	94	**88**	**87**	**78**	**86**	**83**	**75**

Table 5.13: Experimental results with $m = 8, w = 2$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	37	38	19	**33**	**36**	**16**	**32**	**34**	**15**
1.0	65	63	45	**61**	**61**	**41**	**57**	**57**	**38**
1.5	83	76	65	**82**	**75**	**63**	**76**	**73**	**62**
2.0	89	88	80	**88**	**87**	**78**	**81**	**81**	**73**

Table 5.14: Experimental results with $m = 8, w = 4$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	45	41	24	**38**	**38**	**18**	**36**	**35**	**17**
1.0	69	65	49	**64**	**62**	**42**	**60**	**58**	**40**
1.5	85	78	69	**82**	**75**	**63**	**78**	**70**	**59**
2.0	91	90	83	**88**	**87**	**78**	**81**	**81**	**73**

Table 5.15: Experimental results with $m = 8, w = 8$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	49	45	32	**42**	**38**	**18**	**40**	**37**	**18**
1.0	73	68	56	**66**	**62**	**43**	**61**	**58**	**42**
1.5	89	82	76	**82**	**75**	**63**	**81**	**74**	**62**
2.0	93	92	87	**88**	**87**	**78**	**83**	**82**	**75**

Table 5.16: Experimental results with $m = 8, w = 16$

d	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
0.5	57	52	44	**42**	**38**	**18**	**41**	**37**	**18**
1.0	80	74	66	**66**	**62**	**43**	**63**	**59**	**41**
1.5	94	87	84	**82**	**75**	**63**	**78**	**72**	**61**
2.0	97	96	94	**88**	**87**	**78**	**85**	**81**	**73**

Table 5.17: Experimental results of concatenation, union and product

ops	F-KQ	B-KQ	FB-KQ	F-BF	B-BF	FB-BF	F-FW	B-FW	FB-FW
con.	132	138	107	**126**	**132**	**86**	**123**	**129**	**79**
uni.	133	140	99	**125**	**130**	**83**	**120**	**127**	**77**
pro.	137	157	72	**106**	**113**	**22**	**97**	**106**	**21**

Chapter 6

Conclusions and Future Work

In this thesis, we have introduced the notion of "weight redistribution" by investigating the algebraic properties along with the graphical structure inside weighted finite automata. We also generalized the concept of shortest-path problem to finding infimum along every paths and gave an effcient algorithm to solve it. Our algorithm to compute "weight redistribution" is not only the central role in our minimization algorithm, but also applicable on determining the equivalence Z-automata. We also gave two new algorithms, one of which runs in $O(N^3)$, to shrink the state space. These algorithms greatly outperform pervious results in terms of state space.

One possible future work is to improve the time bound which is needed to compute the infimum along every paths. In fact, although Dijkstra's algorithm can not be applied, this is a generalized single-source shortest path problem. It is very likely that this time bound can be reduced by modifying the Floyd-Warshall algorithm.

Another possible future work is that, the assumption of the semirings, that is cancellative and pure, may have much to do with theoratical insterests. By assuming

a cancellative and pure semiring it is possible to find some satisfying properties. Also that we may consider wether deciding the minimality of a deterministic weighted finite automaton is possible.

In section 4.1, it is suggested that the bisimulation relation can be extended to more than one step. In the definition of bisimulation relation, state p and state q are said to be bisimilar if and only if their outgoing transition are the same. We may relax this requirement to something like that p and q are said to be n-bisimilar if and only if their successors after n steps are the same. Of course their behavior must be the same within n steps that can be checked in constant time if n is a constant. In this more general setting, the classical bisimulation relation is the special case of n-bisimulation. To be more precise, it is 0-bisimulation.

Also that we may discover the parallel structure and merge states without transforming them into the trivial cases in the first step of algorithm Minimize-WFA. Along this way, the initial partition is also decided by every states' final weights but it is slightly different that only zero and non-zero are distinguished. This method may makes the state space smaller. Although this method tends to cover the function of weight pushing, algorithm Weights-Redistribution-Efficiently and Generalized-Floyd-Warshall still have their rights due to the example shown in 3.6.

The notion of K-quotient is extend to weighted tree automata in [26]. One possible future work is to extend our works to weighted tree automata, too.

Bibliography

[1] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, R.C. Carrasco, Probabilistic Finite-State Machines–Part I, IEEE Trans. Pattern Analysis and Machine Intelligence, special issue on syntactic and structural pattern recognition, vol 27, no. 7, pp. 1013-1025, 2005.

[2] A. Paz. Introduction to Probabilistic Automata. Academic Press. New York, NY, 1971.

[3] F. Jelinek. Statistical Methods for Speech Recognition. Cambridge, Massachusetts: The MIT Press, 1998.

[4] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method, ser. Lecture Notes in Computer Science, R. C. Carrasco and J. Oncina, Eds., no. 862. Berlin, Heidelberg: Springer Verlag, 1994, pp. 139-150.

[5] L. Saul and F. Pereira. Aggregate and mixed-order Markov models for statistical language processing. In Proceedings of the Second Conference on Empirical Methods in Natural Language Processing, C. Cardie and R. Weischedel, Eds.

Somerset, New Jersey: Association for Computational Linguistics, 1997, pp. 81-89.

[6] D. Ron, Y. Singer, and N. Tishby. Learning probabilistic automata with variable memory length. In Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory. New Brunswick, New Jersey: ACM Press, 1994, pp. 35-46.

[7] J. E. Hopcroft, R. Motwani, J. D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2001.

[8] J. E. Hopcroft. An nlogn algorithm for minimizing the states in a finite automaton. In The Theory of Machines and Computations (Z. Kohavi, ed.), pp. 189-196, Acdemic Press, New York, 1971.

[9] G. Gramlich and G. Schnitger. Minimizing nfa's and regular expressions. In STACS, volume 3404 of Lecture Notes in Computer Science. Springer, 2005.

[10] J. A. Brzozowski. Canonical regular expressions and minimal state staphs for definite events. In Mathematical theory of Automata, Vol. 12 of MRI Symposia Series, pp.529-561, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y. 1962.

[11] J. Eisner. Simpler and More General Minimization for Weighted Finite-State Automata. In Proceedings of HLT-NAACL, Main Papers, pp.64-71. 2003

[12] M. Mohri. Finite-state transducers in language and speech processing. Computational Linguistics, 23(2). 1997.

[13] M. Mohri. Minimization algorithms for sequential transducers. Theoretical Computer Since, 324:177-201. 2000.

[14] R. Gentilini, C. Piazza, A. Policriti. From Bisimulation to Simulation: Coarsest Partition Problems. In Journal of Automated Reasoning 31: 73-103. Kluwer Academic Publishers. 2003.

[15] P. Buchholz. Bisimulation relations for weighted automata. Theoretical Computer Since, 393:109-123. 2008.

[16] D. Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. Int. J. Algebra Comput., 4(3):405-425, 1994.

[17] D. Bustan, O. Grumberg. Simulation-Based Minimization. In ACM Transactions on Computational Logic, Vol. 4, No. 2, April 2003, pp.181-206. 2003.

[18] M. P. Beal, S. Lombardy, J. Sakarovitch. On the Equivalence of Z-Automata. In International Colloquium on Automata, Languages and Programming, LNCS 3580, pp. 397-409, Springer-Verlag Berlin Heidelberg. 2005.

[19] K. Fisler, M. Y. Vardi. Bisimulation Minimization in an Automata-Theoretic Verification Framework. In International Conference on Formal Methods in Computer-Aided Design, LNCS 1522, pp. 115-132, Springer-Verlag Berlin Heidelberg. 1998.

[20] T. Claveirole, S. Lombardy, S. Connor, L. N. Pouchet, J. Sakarovitch. Inside Vaucanson. In International Conference on Implementation and Application of Automata, LNCS 3845, pp. 116-128, Springer-Verlag Berlin Heidelberg. 2006.

[21] S. Lombardy, Y. Regis-Gianas, J. Sakarovitch. Introducing Vaucanson. In International Conference on Implementation and Application of Automata, Theoratical Computer Science 328, pp. 77-96. Elsevier B.V. 2004.

[22] W. Kuich, K. Walk. Block-stochastic matrices and associated finite-state languages. Computing 1, 50-61. 1966.

[23] Vaucanson. http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/Vaucanson

[24] FSM. https://research.att.com/ fsmtools/fsm/

[25] FSA. http://www-i6.informatik.rwth-aachen.de/ kanthak/fsa.html

[26] J. Hogberg, A. Maletti, J. May. Bisimulation Minimisation for Weighted Tree Automata. DLT 2007, LNCS 4588, pp. 229-241, 2007.

[27] L. van Zijl et al. http://www.cs.sun.ac.za/ lynette/MERLin/MerlinManRev.ps, Merlin 1.1 help. Technical report.

[28] J.-M. Champarnaud, G. Hansel, T. Paranthoen, Ziadi. Random generation models for NFAs.J. of Automata, Languages and Combinatorics, 9(2), 2004.