SUCCINCTNESS ANALYSIS OF FINITE AUTOMATON MODELS

by

Emre Erdoğan

B.S., Computer Engineering, Boğaziçi University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2017

SUCCINCTNESS ANALYSIS OF FINITE AUTOMATON MODELS

APPROVED BY:

Prof. Ahmet Celal Cem Say .....
(Thesis Supervisor)

Assoc. Prof. Ali Taylan Cemgil .....

Assoc. Prof. Flavio D'Alessandro .....

DATE OF APPROVAL: 05.07.2017

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my thesis advisor, Prof. Ahmet Celal Cem Say for his advice, guidance, and encouragement, and for introducing me to this subject.

I would also like to thank Assoc. Prof. Flavio D'Alessandro and Assoc. Prof. Ali Taylan Cemgil for agreeing to join my thesis commitee and showing interest in my work.

I would also like to give special mention to Abuzer Yakaryılmaz for kindly taking the time to help me understand various papers I discuss in my thesis.

I will always be indebted to my family and friends for the belief they have shown in me and for their continued support throughout my academic career.

ABSTRACT

SUCCINCTNESS ANALYSIS OF FINITE AUTOMATON MODELS

Finite state automaton, or finite automaton, is a mathematical model of computation and has been one of the most studied models in automata theory. Throughout the years, many different types of finite state automata are proposed, such as deterministic, nondeterministic, probabilistic, and quantum automata. Furthermore, the important questions that how they are related to each other, and how they are related to formal languages, have been a subject of intensive research.

In this thesis, we study the succinctness properties of various finite automata. First, we thoroughly study the topic of simulating various finite automata by deterministic finite automata. Second, we work with three different families of regular languages and we provide the various minimal automata (i.e. minimal in the sense of the number of states used) deciding them. Third, we provide a descriptive form called "Unary Finite Periodic Form", or shortly UFPF, to efficiently describe regular languages over unary alphabets and we introduce algorithms to show the efficient realization of closure properties of UFPF.

ÖZET

SONLU ÖZDEVİNİR MODELLERİNDE ÖZLÜLÜK ANALİZİ

Sonlu özdevinir ya da sonlu durum makinesi, bilgisayımda kullanılan matematiksel bir modeldir ve özdevinir teorisinde en çok çalışılan modellerden biridir. Yıllar boyunca deterministik, nondeterministik, olasılıksal ve kuantum özdevinir gibi sonlu durum makinelerinin birçok farklı çeşidi önerilmiştir. Ayrıca, bu özdevinir modellerinin birbirleri arasındaki ilişkileri ve biçimsel dillerle olan bağları üzerine ayrıntılı çalışmalar yapılmıştır.

Bu tezde, özdevinir modellerinde özlülük özelliği üzerine çözümlemeler yapılmıştır. Konu kapsamı, üç ana başlık altında ayrıntılandırılmıştır. İlk olarak, çeşitli sonlu makine modellerinin deterministik özdevinirler tarafından simüle edilmesi konusunda yapılan çalışmalar yer almıştır. İkinci olarak, üç düzenli dil ailesi tanımlanmış ve bu dil ailelerinin farklı özdevinir modelleri tarafından en az kaç durum ile tanınacağı gösterilmiştir. Üçüncü olarak, tek harfli alfabeler kullanılarak yaratılabilecek düzenli dilleri ve bu dilleri tanıyan sonlu makineleri birlikte tanımlayan bir biçim geliştirilmiştir. Adı, "Tekli Sonlu Periyodik Biçim" olan bu form kullanılarak düzenli dillerin kapalılık özellikleri üzerine detaylıca çalışılmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathcal{P}(Q)$	Power set of Q
q_i	i^{th} state
Q	Set of states
$\mathbb{Q}$	Set of rationals
$\mathbb{R}$	Set of real numbers
$\mathbb{Z}$	Set of integers
δ	Transition function
Σ	Alphabet
Σ^*	The set of all strings written by the symbols in Σ

LIST OF ACRONYMS/ABBREVIATIONS

2AFA	2-way alternating finite automata
2DFA	2-way deterministic finite automata
2NFA	2-way nondeterministic finite automata
2PFA	2-way probabilistic finite automata
2QFA	2-way quantum finite automata
AFA	Alternating finite automata
CDNF	Canonical disjunctive normal form
$CDNF_S$	Set of all possible CDNFs that can be generated from a set S of variables
CNF	Conjunctive normal form
CUFPF	Compact unary finite periodic form
DFA	Deterministic finite automata
DNF	Disjunctive normal form
gcd	greatest common divisor
lcm	least common multiple
max	maximum
min	minimum
NFA	Nondeterministic finite automata
OQFA	Orthogonal quantum finite automata
pDFA	Partial deterministic finite automata
PFA	Probabilistic finite automata
QFA	Quantum finite automata
UFPF	Unary finite periodic form

1. INTRODUCTION

In automata theory, a finite automaton, or finite state automaton, is an abstract, mathematical model of computation which also has roles in several applied areas of computer science. The most basic version of finite state automata is called "deterministic finite automaton", or shortly DFA, and it can be considered as the heart of automata theory. Throughout the years, other than DFA, different versions of finite state automata are proposed by various scientists by analyzing different notions' relation to computation, such as nondeterminism, probability, and quantum, and the important question that how they are related to each other, and how they are related to formal languages, has been a subject of intensive research [1–14].

The main focus of concept in this work, namely "succinctness", is a characteristic of speech, writing, and thought in general that shows both clarity and brevity. For example, consider two languages L_1 and L_2. L_1 is the language of words such that length of any word is a multiple of 10. L_2 is the language of words such that length of any word is either a multiple of 10 or a multiple of 20. One can easily see that both languages are equal, yet L_1's description is much simpler or much succinct than L_2's description, so we naturally prefer the former when we talk about them.

In the context of descriptional complexity of finite state automata, we mention succinctness as a property of automata, and it is related to the following question: "Given a regular language, at least how many states a given type of automaton needs in order to recognize it?"

In addition to that, computer scientists are greatly interested in the relative succinctness property of different kinds of finite automata, and that is related to the following questions: "Can an X-type finite automaton perform statewise better than (i.e. with fewer states) Y-type finite automaton when recognizing a particular language? For which families of languages X-type performs best against Y-type?"

To solve these questions, much effort has been put, and in this body of work, we want to analyze the solutions given for them and further contribute when it is possible.

After this introduction chapter, we give preliminary information about the main topics we discuss in Chapter 2. This part mainly consists of definitions and basic theorems that will be useful throughout this particular work. In Chapter 3, we study the topic of simulating various finite automata (e.g. nondeterministic finite automata, probabilistic finite automata) by DFA. For various kinds of automata, we give thorough answers to two important questions here:

- To simulate an X-type finite automaton, or XFA, at most how many states a DFA needs?
- Does there exist a regular language that to simulate the deciding XFA, a DFA needs at least as many states as a DFA needs at most for a general simulation?

In Chapter 4, we study three interesting families of unary and binary regular languages. For every finite automaton type that we use throughout this work, we provide the minimal automata (i.e. minimal in the sense of the number of states used) that decide those families of languages and prove why and how they do that. When we cannot provide such a strong argument, we try to show if some automaton type can be statewise advantageous against another types of automata, especially against DFA, when deciding a particular family of languages. After this part, in Chapter 5, we provide a descriptive form called Unary Finite Periodic Form, or shortly UFPF, to efficiently describe regular languages over unary alphabets (i.e. alphabets consisting of one symbol). Moreover, we analyze closure properties of URLs and we introduce algorithms to show the efficient realization of closure properties of UFPF. Finally, we conclude our study with open problems and future work in the sixth chapter.

2. PRELIMINARIES

In this part, we state the basic concepts that will be useful in the following chapters. We begin with definitions of words, languages and various automata. Definitions and theorems are largely cited from Michael Sipser's well-known book "Introduction to the Theory of Computation" [15] and Christos Kapoutsis' PhD thesis "Algorithms and lower bounds for finite automata size complexity" [16], unless stated otherwise.

Definition 2.1. An *alphabet* Σ is a finite set of symbols or letters.

Definition 2.2. A *word* w *over alphabet* Σ is a finite sequence of zero or more *symbols* of Σ. Σ^* denotes the set of all words over A. The *empty word* is denoted as ϵ.

Definition 2.3. The *extended word* w_e *of* w *over* Σ is written as $w_e = \$w\#$ where $\$, \# \notin \Sigma$.

Definition 2.4. The *length of* w is a non-negative integer denoted by $|w|$ such that if $w = \epsilon$, then $|w| = 0$, else for $w = w_1 w_2 ... w_k$, where w_is are letters, $|w| = k$.

Definition 2.5. f is a *factor of* w if there exists λ, μ such that $w = \lambda f \mu$. p is a *prefix of* w if there exists μ such that $w = p\mu$. s is a *suffix of* w if there exists λ such that $w = \lambda s$.

Definition 2.6. A *language* L *over alphabet* Σ is a set of words w where every $w \in \Sigma^*$. L can be finite or infinite.

Definition 2.7. For a word $w = w_1 w_2 ... w_k$, where w_is are letters, the *reverse of* w is $w^R = w_k ... w_2 w_1$, the word w in reverse order. For a language L, *the reverse of* L is $L^R = \{w^R \mid w \in L\}$.

Definition 2.8. A *deterministic finite automaton*, or *DFA*, is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite set called the *states*,

(ii) Σ is a finite set called the *alphabet*,

(iii) $\delta : Q \times \Sigma \to Q$ is the *transition function*,

(iv) $q_0 \in Q$ is the *start state*, and

(v) $F \subseteq Q$ is a set called the *accepting (final) states*.

It is inferred that $F' = Q - F$ is the set of *rejecting states*. Also, the transition function of DFA is *complete*, meaning that there is *one* transition for each state and input symbol pair.

Visually speaking, the states and transition function of DFA can be represented as directed graphs where states are vertices and transitions are directed edges or arrows. In the following chapters, we will use these more visualizable concepts in other types of automata as well.

Definition 2.9. A *partial DFA*, shortly *pDFA*, is a DFA that has a partial transition function, meaning that there is at most one transition for each state and input symbol pair. A pDFA halts and rejects on an undefined transition.

Definition 2.10. The *computation c of DFA D on word w* is the unique sequence of states $c = r_0, r_1, ..., r_k$ such that

(i) $r_i \in Q$ for $i = 0, ..., k$,

(ii) $r_0 = q_0$ (the start state),

(iii) $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, ..., k-1$,

where $w = w_1 w_2 ... w_k$ (w is composed of k symbols). c is an *accepting computation* if $r_k \in F$.

Definition 2.11. A DFA D *accepts* a word w if its computation on w is an accepting computation.

Definition 2.12. A DFA D *decides (recognizes) language L* if $L = \{w \mid D \text{ accepts } w\}$.

Definition 2.13. A language is called a *regular language* if some DFA recognizes it.

Definition 2.14. For an alphabet A, let $L \subseteq A^*$, the right-equivalence relation $\equiv_L$ generated by L is defined as follows. For $x, y \in A^*$, $x \equiv_L y$, which means x *is pairwise equivalent to* y *by* L, if and only if for all $z \in A^*$ we have $xz \in A^*$ if and only if $yz \in A^*$.

Theorem 2.1. Given a language L, if the number of equivalence classes generated by $\equiv_L$ is finite and equals to n, then there exists a DFA with n states that decides L [1].

Definition 2.15. Let L_1 and L_2 be languages. The *regular operations* are defined as follows:

- *Union:* $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$.
- *Concatenation:* $L_1 \circ L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}$.
- *Star:* $L_1^* = \{w_1 w_2 ... w_k \mid k \geq 0 \text{ and each } w_i \in L_1\}$.

Definition 2.16. Let L_1 and L_2 be languages. The *boolean operations* are defined as follows:

- *Union:* $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$.
- *Intersection:* $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ and } w \in L_2\}$.
- *Complement:* $\bar{L}_1 = \Sigma^* \setminus L_1$.

Theorem 2.2. The class of regular languages are closed under the union, intersection, complement, concatenation, and star operations.

Definition 2.17. A *nondeterministic finite automaton*, shortly *NFA*, is a 5-tuple $(Q, \Sigma_\epsilon, \delta, q_0, F)$, where

(i) Q is the set of states,
(ii) Σ_ϵ is the alphabet,
(iii) $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function,
(iv) $q_0 \in Q$ is the start state, and
(v) $F \subseteq Q$ is the accepting states,

where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\mathcal{P}(Q)$ denotes the *power set of* Q. Here, the transition function of NFA is complete.

Definition 2.18. A *computation c of NFA N on word w* is a sequence of states $c = r_0, r_1, ..., r_k$ such that

(i) $r_i \in Q$ for $i = 0, ..., k$,
(ii) $r_0 = q_0$,
(iii) $r_{i+1} \in \delta(r_i, w_{i+1})$ for $i = 0, ..., k-1$

where $w = w_1 w_2 ... w_k$ (w is composed of k symbols). A computation $c = r_0, r_1, ..., r_k$ is an *accepting computation* if $r_k \in F$. One should be careful that in the DFA case there is only one computation on w but in the NFA case there may exist more than one computation on w.

Definition 2.19. A NFA N *accepts* a word w if at least one computation on w is an accepting computation.

Definition 2.20. A *2-way deterministic finite automaton*, shortly denoted as *2DFA*, is a 5-tuple $(Q, \Sigma_{\{\$,\#\}}, \delta, q_0, q_f)$, where

(i) Q is the set of states,
(ii) $\Sigma_{\{\$,\#\}}$ is the alphabet,
(iii) $\delta : Q \times \Sigma_{\{\$,\#\}} \to Q \times \{l, r\}$ is the transition function,
(iv) $q_0 \in Q$ is the start state, and
(v) $q_f \in Q$ is the accepting state,

where $\Sigma_{\{\$,\#\}} = \Sigma \cup \{\$, \#\}$ and the symbols \$, #, l, and r denote the left endmarker, right endmarker, left move, and right move, respectively. The transition function of 2DFA is partial.

Here, the working principle of 2DFA is slightly different than that of DFA. We can think the transition function of DFA as an instruction set to move DFA's "head", which enables DFA to read the input word w character by character, to "right" (the only possible direction for DFA) and change its current state accordingly. For 2DFA, the transition function is there to move 2DFA's head, which enables 2DFA to read the

extended input word $w_e = \$w\#$ character by character, to "right" or "left" and change its current state. As a result, 2DFA can take many more steps than the length of the input to accept/reject it. Also, δ obeys some additional rules: On the left endmarker it either moves its head to the right or hangs (stops on some non-accepting state) and on the right endmarker it either moves the head to the left, hangs, or moves the head to the right and enters q_f. We will use this "head" concept in other 2-way automata descriptions.

Definition 2.21. The *m-step computation c of 2DFA D when started at state q on the i^{th} symbol of w*, or shortly $COMP_{D,q,i}(w, m)$, is the unique sequence of state-position pairs $c = (r_0, i_0), (r_1, i_1), ..., (r_m, i_m)$ such that

(i) $r_j \in Q$ for $j = 0, ..., m$,
(ii) $i_j \in \{1, 2, ..., |w|\}$ for $j = 0, ..., m$,
(iii) $(r_0, i_0) = (q, i)$,
(iv) $\delta(r_j, \sigma) = (r_{j+1}, d)$ for $j = 0, ..., m-1$ and $((d = l \ \wedge \ i_{j+1} = i_j - 1) \ \vee \ (d = r \ \wedge \ i_{j+1} = i_j + 1))$,

where $w = w_1 w_2 ... w_k$. $COMP_{D,q,i}(w, m)$ can be shortened to $COMP_{D,q,i}(w)$ if the number of steps taken m is not important. $COMP_{D,q_0,1}(\$w\#)$ is called a *natural computation*. A natural computation is *accepting* if $(r_m, i_m) = (q_f, |\$w\#| + 1)$.

Definition 2.22. A 2DFA *accepts* a word w if the natural computation on $\$w\#$ is accepting.

Definition 2.23. A *2-way nondeterministic finite automaton*, or shortly *2NFA*, is a 5-tuple $(Q, \Sigma_{\{\$,\#\}}, \delta, q_0, q_f)$, where

(i) Q is the finite set of states,
(ii) $\Sigma_{\{\$,\#\}}$ is the finite alphabet,
(iii) $\delta : Q \times \Sigma_{\{\$,\#\}} \to \mathcal{P}(Q \times \{l, r\})$ is the transition function,
(iv) $q_0 \in Q$ is the start state, and
(v) $q_f \in Q$ is the accepting state,

where $\Sigma_{\{\$,\#\}} = \Sigma \cup \{\$, \#\}$ and $\$$, $\#$, l, and r denote the left endmarker, right endmarker, left move, and right move respectively. The transition function of 2NFA is complete.

Definition 2.24. An *m-step computation c of 2NFA N when started at state q on the i^{th} symbol of w*, or shortly $COMP_{N,q,i}(w, m)$, is a sequence of state-position pairs $c = (r_0, i_0), (r_1, i_1), ..., (r_m, i_m)$ such that

(i) $r_j \in Q$ for $j = 0, ..., m$,
(ii) $i_j \in \{1, 2, ..., |w|\}$ for $j = 0, ..., m$,
(iii) $(r_0, i_0) = (q, i)$,
(iv) $\delta(r_j, \sigma) \supseteq \{(r_{j+1}, d)\}$ for $j = 0, ..., m-1$ and $((d = l$ and $i_{j+1} = i_j - 1)$ or $(d = r$ and $i_{j+1} = i_j + 1))$,

where $w = w_1 w_2 ... w_k$. $COMP_{D,q,i}(w, m)$ can be shortened to $COMP_{D,q,i}(w)$ if the number of steps taken m is not important. $COMP_{N,q_0,1}(\$w\#)$ is called a *natural computation*. A natural computation is *accepting* if $(r_m, i_m) = (q_f, |\$w\#| + 1)$. Clearly, the definitions are very similar to those of 2DFA, the only difference is that a computation may not be unique in 2NFA case (as is in the NFA-DFA comparison).

Definition 2.25. A 2NFA *accepts* a word w if at least one natural computation on $\$w\#$ is accepting.

In the literature, the concept of "alternating finite automata" is defined differently by various authors in [4–7, 17, 18]. For this kind of automaton, we will mainly use the following definition which is very similar to the one in Brzozowski and Leiss' paper "On equations for regular languages, finite automata, and sequential networks" (1980) [6].

Definition 2.26. A *truth table* is a mathematical table to compute the functional values of logical expressions on each of their functional arguments, that is, on each combination of values taken by their logical variables [19].

Definition 2.27. A formula of n boolean variables $x_1, ..., x_n$ is in *disjunctive normal form*, shortly DNF, if it is a disjunction of 0 or more *clauses*, where a clause consists of a conjunction of those variables (i.e. x_i) or their complement variables (i.e. $\bar{x}_i$). A

canonical disjunctive normal form, shortly CDNF, is a DNF that obeys the following rules:

(i) Every clause is a conjunction of n components.
(ii) In a clause, either some variable x_i or its complement $\bar{x}_i$ appears only once.

The set of all possible different CDNFs that can be generated from the set S of n variables is called the *all-CDNF set of* S, shortly shown as $CDNF_S$. One can see that for any set S, $1 \in CDNF_S$ and $0 \in CDNF_S$ (i.e. "tautology" and "contradiction", respectively). Additionally, one may want to note that we only say "a CDNF" instead of "a formula in CDNF" (as an abbreviation).

Definition 2.28. An *alternating finite automaton* is a 5-tuple $(Q, \Sigma, \delta, f_0, F)$, where

(i) Q is the set of states,
(ii) Σ is the finite alphabet,
(iii) $\delta : Q \times \Sigma \to CDNF_Q$ is the transition function,
(iv) $f_0 \in CDNF_Q$ is the starting formula, and
(v) $F \subseteq Q$ is the set of final states,

where $CDNF_Q$ is the all-CDNF set of Q. Q is also regarded as a set of n boolean variables where each state-variable is associated with the value 1 if it is a final state (i.e. "true") or 0 (i.e. "false") if it is not a final state. δ gives a propositional formula in $CDNF_Q$ for each state-symbol pair. Alternating finite automata are shortly called as *AFA*.

Here, it is implicitly given that for $Q = \{q_1, q_2, ..., q_p\}$, $\bar{Q} = \{\bar{q}_1, \bar{q}_2, ..., \bar{q}_p\}$. Furthermore, for every $q \in Q$ such that $\delta(q, a) = f$, $\delta(\bar{q}, a) = \bar{f}$. Now, we give an extended version of the transition function δ of AFA:

Definition 2.29. Given an AFA $(Q, \Sigma, \delta, f_0, F)$ and $|Q| = p$, there is an extended version of δ, named as δ_e such that for any $f \in CDNF_Q$ given as $f = (v_1^1 \wedge ... \wedge v_p^1) \vee$

$(v_1^2 \wedge ... \wedge v_p^2) \vee ... \vee (v_1^k \wedge ... \wedge v_p^k)$:

$$\delta_e(f,a) = (\delta(v_1^1,a) \wedge ... \wedge \delta(v_p^1,a)) \vee (\delta(v_1^2,a) \wedge ... \wedge \delta(v_p^2,a)) \vee ... \vee (\delta(v_1^k,a) \wedge ... \wedge \delta(v_p^k,a))$$

where $v_i^j \in Q$ or $v_i^j \in \bar{Q}$ for $1 \leq i \leq p$ and $1 \leq j \leq k$, for some given symbol a. Surely, there is a CDNF that is equivalent to $\delta_e(f,a)$ and it can be calculated from $\delta_e(f,a)$ by using logical equivalences.

As the input is read, the automaton builds a propositional formula in CDNF, and on reading some symbol a, replaces every q in the current formula by $\delta(q,a)$ (as defined just above in the definition of δ_e), and shorten the formula into CDNF. An AFA A *accepts* a word w if the final formula evaluates 1 (i.e. true) when 1 is substituted for every $q \in F$ in the final formula and 0 (i.e. false) is substituted for every $q \notin F$ in the final formula.

Now we define "2-way alternating finite automata" concept. We will not use the definitions given in [5] or [13], instead we will use a 2-way and modified version of the definition 2.28.

Definition 2.30. A *2-way alternating finite automaton* is a 5-tuple $(Q, \Sigma_{\{\$,\#\}}, \delta, f_0, F)$, where

(i) Q is the finite set of states,
(ii) $\Sigma_{\{\$,\#\}}$ is the finite alphabet,
(iii) $\delta : Q \times \Sigma_{\{\$,\#\}} \to CDNF_Q \times \{l, r\}$ is the transition function,
(iv) $f_0 \in CDNF_Q$ is the starting formula, and
(v) $F \subseteq Q$ is the set of final states,

where $\Sigma_{\{\$,\#\}} = \Sigma \cup \{\$, \#\}$ and $\$, \#, l, r$ respectively denote the left endmarker, right endmarker, left move, and right move, and $CDNF_Q$ is the all-CDNF set of Q. Similar to the AFA definition, Q is also regarded as a set of n boolean variables where each state-variable is associated with the value 1 if it is a final state (i.e. "true") or 0 (i.e.

"false") if it is not a final state. δ gives a propositional formula in $CDNF_Q$ and a left or right move for each state-symbol pair. 2-way alternating finite automata are shortly called as *2AFA*.

Here, it is implicitly given that for $Q = \{q_1, q_2, ..., q_p\}$, $\bar{Q} = \{\bar{q}_1, \bar{q}_2, ..., \bar{q}_p\}$. Furthermore, for some $q \in Q$ such that $\delta(q, a) = (f, d)$, $\delta(\bar{q}, a) = (\bar{f}, d)$.

Definition 2.31. Say a 2AFA $(Q, \Sigma_{\{\$,\#\}}, \delta, f_0, F)$, a word $w = w_1 w_2 ... w_r$, and $n = |Q|$ is given where all such $w_i \in \Sigma$. An *m-step computation tree of 2AFA D when started at CDNF formula h on the p^{th} symbol of w*, or shortly $COMP_{D,h,p}(w, m)$, is a tree of nodes such that

(i) Every node is a CDNF-position pair.

(ii) The *root* node is the starting node (h, p).

(iii) A node (f, x) is called *terminal* if $x = |w| + 1$ or $x = 0$.

(iv) Every non-terminal node has n *child* nodes, defined by the following:
Say (f, x) is a node such that $f = (f_1^1 \wedge ... \wedge f_n^1) \vee (f_1^2 \wedge ... \wedge f_n^2) \vee ... \vee (f_1^k \wedge ... \wedge f_n^k)$ where $f_i^j \in Q$ or $f_i^j \in \bar{Q}$ for $1 \leq i \leq p$ and $1 \leq j \leq k$. (f, x) has n *child* nodes (g_i^l, y_i^l) for $1 \leq i \leq n$ such that:
Take a conjunction inside f, say $(f_1^l \wedge ... \wedge f_n^l)$ for some fixed $l \leq k$. For every $i \leq n$, if $\delta(f_i^l, w_x) = (g, d)$, then $g_i^l = g$ and $y_i^l = x + 1$ if $d = r$, or $y_i^l = x - 1$ if $d = l$, where w_x denotes the symbol on the x^{th} position of w.
One can see that every non-terminal node (f, x) in this form can have k different sets of n child nodes.

(v) The depth of $COMP_{D,h,i}(w, m)$ is m.

If the number of steps taken m is not important, then $COMP_{D,h,i}(w, m)$ can be shortened to $COMP_{D,h,i}(w)$. For a given computation tree T that has the root node (f, x), a *computation subtree T' of T* is a computation tree that has a child of (f, x) as its root node and all of its descendants in T. $COMP_{D,f_0,1}(\$w\#)$ is called a *natural computation tree*. A natural computation tree T is *accepting* if:

- Every computation subtree of T is also accepting.
- If a computation subtree T' is $T' = (g, |\$w\#| + 1)$ and f evaluates 1 (i.e. true) when 1 is substituted for every $q \in F$ in g and 0 (i.e. false) is substituted for every $q \notin F$ in g, then T' is accepting.

Definition 2.32. A 2AFA *accepts* a word w if a natural computation tree on $\$w\#$ is accepting.

Now we define probabilistic finite automata concept. We use a slightly modified version of the definitions given in Rabin's well-known paper "Probabilistic Automata" (1963) [3].

Definition 2.33. A *probabilistic finite automaton* is a 5-tuple (Q, Σ, M, q_s, F), where

(i) $Q = \{q_1, ...q_n\}$ is the set of states,
(ii) Σ is the alphabet,
(iii) $M = \{M_\sigma \mid \sigma \in \Sigma\}$ is a finite collection of $|Q| \times |Q|$-dimensional matrices M_σ called the *transition probabilities matrices (tables)*,
(iv) $q_1 \in Q$ is the start state, and
(v) $F \subseteq Q$ is the set of accepting states.

Here, an entry $M_\sigma[i, j]$ in M_σ (i.e. entry in the i^{th} row and j^{th} column of M_σ) is called the *transition probability from state q_j to q_i on symbol σ*. Transition probabilities are real numbers in $[0, 1]$. Every M_σ is a *column stochastic matrix* which means that for every column, entries in a column adds up to 1. In other words, we can say that for any column j of M_σ:

$$0 \leq M_\sigma[i, j] \leq 1, \ \sum_{i=1}^{n} M_\sigma[i, j] = 1 \tag{2.1}$$

A PFA P reads as follows: Given some word w, P starts its computation in the starting state. When in state q_j, if it reads a symbol σ, it goes into the state q_i with probability $M_\sigma[i, j]$. At the end of its computation, P is in the state q_i with some

probability p_i. For $q_i \in F$, such p_i are summed to find the acceptance probability of w by P.

Last but not least, a PFA does not always have to start with only one starting state. For example, it can start on some state q_i with probability p_i, where the sum of all such p_i should add up to 1. For the sake of simplicity, we generally say that it starts on the starting state q_1 with probability 1.

Probabilistic finite automata are shortly called as *PFA*.

Definition 2.34. Suppose some word $w = w_1w_2...w_r$ where $|w| = r$ is given and every w_i is a symbol where $1 \leq i \leq r$. The *computation* ρ_w *of PFA* P *on word* w is the $n \times 1$-dimensional matrix resulted after reading w and recursively defined as the following matrix product:

$$\rho_{uw_j} = M_{w_j}\rho_u \tag{2.2}$$

where $u = w_1...w_{j-1}$ and ρ_ϵ is the starting probability distribution and conventionally, $\rho_\epsilon = (1 \;\; 0 \;\; ... \;\; 0)^T$ (i.e. at the beginning, we are in the starting state with probability 1). $\rho_w[l]$ (i.e. the l^{th} entry of ρ_w) denotes the *probability of* P *being in state* q_l *after reading* w. ρ_w is also called as the *final probability distribution vector of* w.

Definition 2.35. A PFA P *accepts* a word w with probability $p(w) = \sum_{q_i \in F} \rho_w[i]$ where F and $\rho_w[i]$ respectively denote the set of final states of P and i^{th} entry of ρ_w.

Definition 2.36. A language L is recognized by a *bounded-error PFA with error bound* ϵ where $0 \leq \epsilon < 0.5$ if $p(w) \geq 1 - \epsilon$ for any $w \in L$ and $p(w) \leq \epsilon$ for any $w \notin L$.

Bounded-error PFA are equivalent to DFA in language recognition power, a feature which we will prove later. Now, we define the concept of quantum finite automata, shortly *QFA*. Throughout the years, many different QFA definitions have been proposed (a list can be seen in [14]). We will use the most generalized version of these QFA, mostly utilizing the definitions given in [14] and [20].

Definition 2.37. A *quantum finite automaton* is a 5-tuple $(Q, \Sigma, \{\epsilon_\sigma\}_{\sigma\in\Sigma}, q_1, F)$, where

(i) $Q = \{q_1, q_2, ..., q_n\}$ is the set of states,
(ii) Σ is the alphabet,
(iii) $\{\varepsilon_\sigma\}_{\sigma\in\Sigma}$ is a finite collection of $|Q|m \times |Q|$-dimensional matrices (i.e. rectangles) called *superoperators*, each of which are composed of $|Q| \times |Q|$-dimensional matrices called the *operators* $\{E_{\sigma,i} \mid 1 < i < m\}$ for a positive integer m,
(iv) $q_1 \in Q$ is the start state, and
(v) $F \subseteq Q$ is the accepting states.

There exists three rules for numbers in those superoperators $\{\varepsilon_\sigma\}_{\sigma\in\Sigma}$

(i) All entries of a superoperator are real numbers in the interval $[-1, 1]$,
(ii) Euclidean norm of each column of a superoperator should be equal to 1, and
(iii) Every pair of different columns should be orthogonal to each other.

If $m = 1$, then superoperators become orthogonal matrices with dimensions of $|Q| \times |Q|$. In the literature, any entry α of a superoperator can be a complex number such that $|\alpha| \leq 1$. To keep our model simple, we will use real numbers instead of complex numbers.

Definition 2.38. Suppose some word $w = w_1w_2...w_r$ where $|w| = r$ is given and every w_i is a symbol where $1 \leq i \leq r$. The *computation* ρ_w *of QFA* P *on word* w is a matrix of size $|Q| \times |Q|$ resulted after reading w and recursively defined as

$$\rho_{uw_j} = \sum_i E_{w_j,i}\rho_u E^\dagger_{w_j,i} \tag{2.3}$$

where $u = w_1...w_{j-1}$ and ρ_ϵ is the starting quantum distribution matrix and conventionally, a $|Q| \times |Q|$-dimensional single-entry matrix in which the entry in the first column and first row is 1. $E^\dagger$ denotes the transpose of E. ρ_w is also called as the *final quantum distribution matrix of* w.

Definition 2.39. A QFA P *accepts* some word w with probability $p(w) = \sum_{q_i \in F} \rho_w[i][i]$ where F and $\rho_w[i][i]$ respectively denote the set of final states of P and the entry in the i^{th} row and i^{th} column of ρ_w.

In the literature, the matrices ρ_w are called the density matrices, ρ_ϵ being the initial density matrix. There are two important properties of density matrices:

(i) Numbers on the diagonal of a density matrix are always real numbers (even if we choose our superoperators to have complex numbers instead of real numbers),
(ii) i^{th} diagonal entry $d_{i,i}$ of a density matrix represents the probability of being in state q_i,
(iii) Sum of the diagonal elements (i.e. the trace) of a density matrix is 1.

Using density matrices with superoperators as defined above, this representation can be regarded as a way of "mathematizing" how quantum finite automata behave (i.e. the finite automata that work with principles governed by quantum mechanics). By that, we can represent the superposition of states (i.e. the circumstance of being in more than one state at the same time) and the state transitions that happen in the quantum world in a simple way.

A QFA P reads a word as follows: Given a word $w = w_1w_2...w_r$, P starts its computation as the given matrix ρ_ϵ which is a way of saying that it starts on the starting state. In the j^{th} step, as P "reads" some symbol w_j, which can be represented as performing a series of matrix operations on the previous computed matrix ρ_u ($u = w_1...w_{j-1}$) with $E_{w_j,i}$ and its conjugate transpose, for all possible such $E_{w_j,i}$. At the end, it accepts w with the probability $p(w) = \sum_{q_i \in F} \rho_w[i][i]$ which is a way of saying that the computation ends on the final states with a total probability of $p(w)$.

As one can see, we try to define QFA as simple and mathematized as possible, as a type of one-way machine being capable of doing operations that can be represented as certain forms of matrix operations. Later, we will explain more about QFA. For

more information about how quantum states behave physically in a working QFA, one can see [14] and [20].

Definition 2.40. A language L is recognized by a *bounded-error QFA with error bound* ϵ where $0 \leq \epsilon < 0.5$ if $p(w) \geq 1 - \epsilon$ for $w \in L$ and $p(w) \leq \epsilon$ for $w \notin L$.

Bounded-error QFA are equivalent to DFA in language recognition power, a feature which we will prove later. Readers may be curious about whether there exists 2-way probabilistic finite automata and 2-way quantum finite automata, shortly *2PFA* and *2QFA*, and the answer is of course "yes". 2PFA and 2QFA are powerful in terms of language recognition power and they recognize languages other than regular, even with bounded-error. For more information, one may want to see the papers [14] and [21].

Here are the other important definitions.

Definition 2.41. The *state complexity of a regular language* is the number of states of the minimal DFA that recognizes it. *Operational state complexity* is the study of the state complexity of operations over languages [22].

Definition 2.42. A DFA (or 2DFA, NFA, 2NFA, AFA, 2AFA, PFA, QFA) M is said to be *minimal* if there does not exist another DFA (resp. 2DFA, NFA, 2NFA, AFA, 2AFA, PFA, QFA) recognizing the same language that M recognizes with fewer number of states.

Definition 2.43. Let f and g be two functions defined on some subset of the real numbers. One writes $f(x) = O(g(x))$ where $x \to \infty$ if and only if there is a positive real number M and a real number x_0 such that $|f(x)| \leq M|g(x)|$ for all $x \geq x_0$.

Definition 2.44. Given m numbers $\{x_1, x_2, ..., x_m\}$, when we say $\max(x_1, x_2, ..., x_m)$ and $\min(x_1, x_2, ..., x_m)$, we mean the *maximum* (or biggest) and *minimum* (or smallest) of all those m numbers, respectively.

Definition 2.45. The *least common multiple* of m numbers $\{x_1, x_2, ..., x_m\}$, denoted by $\text{lcm}(x_1, x_2, ..., x_m)$, is the smallest positive integer that is divisible by all those m numbers. Given two n-bit numbers x_1 and x_2, computing $\text{lcm}(x_1, x_2)$ costs $O(n^2)$ steps of bitwise operations by using well known Euclidean algorithm.

Definition 2.46. The *greatest common divisor* of m numbers $\{x_1, x_2, ..., x_m\}$, denoted by $\gcd(x_1, x_2, ..., x_m)$, is the biggest positive integer that divides all those m numbers. Given two n-bit numbers x_1 and x_2, computing $\gcd(x_1, x_2)$ costs $O(n^2)$ steps of bitwise operations by using well known Euclidean algorithm.

3. SIMULATION OF FINITE AUTOMATA BY DFA

It is widely known that DFA are equivalent to other finite automata, namely NFA, AFA, 2DFA, 2NFA, 2AFA, bounded-error PFA, and bounded-error QFA in language recognition power. However, generally, a DFA needs much more states to recognize a regular language than its counterparts do. In this part, we will provide a thorough analysis on simulating a given type of finite automata by DFA, or pDFA when it is more suitable. We will explain the simulation methods, and the upper bounds and lower bounds for such conversions. An upper bound is generally described with an argument similar to the following one:

"Every X-type finite automaton with n states has an equivalent DFA with at most $F(n)$ states where $F(n)$ is a function of n."

Similarly, a lower bound is generally described with an argument similar to the following one:

"Some X-type finite automaton with n states has an equivalent DFA with at least $G(n)$ states where $G(n)$ is a function of n."

Apart from those, while simulating an X-type finite automaton by DFA, we preemptively ask two questions:

(i) What is the relation between the states of X type finite automaton and those of DFA?
(ii) How to build the transition function of DFA using that of X type?

We will use these guiding questions in our conversions. Now, we begin with simulation of NFA by DFA.

3.1. Simulation of NFA by DFA

For NFA to DFA conversion, the answers to our questions are relatively simple and the proofs are shown before [1], [15]:

(i) The correct mathematical object to relate the states of NFA to those of DFA is the "power set". Meaning, the power set of states of an NFA should be the set of states of the equivalent DFA.

(ii) For some symbol a of NFA's alphabet, take a subset T of NFA's states and group all the states that T's outgoing arrows reach on a. Repeat this for all subsets.

Now, we formally prove the following theorem:

Theorem 3.1. Every NFA with n states has an equivalent DFA with at most 2^n states [15].

Proof. Let NFA $N = (Q, \Sigma_\epsilon, \delta, q_0, F)$ where the size of Q is $|Q| = n$. The equivalent DFA $D_N = (Q', \Sigma', \delta', q_0', F')$ where

- $Q' = \mathcal{P}(Q)$ is the power set of Q,
- $\Sigma' = \Sigma_\epsilon \setminus \{\epsilon\}$
- $\delta'(q', a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \subseteq q'\}$,
- $q_0' = E(\{q_0\})$,
- $F' = \{R \in Q' \mid R \text{ has an accept state of } N\}$.

where $E(R) = \{q \mid q \text{ can be reached from } R \text{ by using 0 or more } \epsilon \text{ arrows}\}$ for $R \subseteq Q$.

Now, we now explain more about δ'. For $q' \in Q'$ and $a \in \Sigma$, take our $\delta'(q', a)$ where q' is both a state of D_N and a set of states of N. For D_N to read the symbol a while on state q', δ' actually shows the rules where a takes q' to. Considering N, every state $r \in q'$ may go to a set of states $\delta(r, a)$ and we include the states that can be reachable from $\delta(r, a)$ by only ϵ arrows, forming $E(\delta(r, a))$. Thus, we take the union

of all these sets to construct δ':

$$\delta'(q', a) = \bigcup_{r \in q'} E(\delta(r, a)) \tag{3.1}$$

This ends the construction of the DFA D_N from the NFA N. Clearly, D_N is equivalent to N and D_N has 2^n states. □

2^n is the upper bound for conversion of NFA into DFA. Now, we will show the lower bound:

Theorem 3.2. Some NFA with $n + 1$ states has an equivalent DFA with at least 2^n states.

Proof. The language we will give is the same as Sipser's [15]. Let C_n be the language of binary strings that has "0" as its n^{th} bit from the end. The NFA N_{C_n} recognizing this language needs $n + 1$ states and $N_{C_n} = (\{q_0, ..., q_n\}, \{0, 1\}_\epsilon, \delta, q_0, \{q_n\})$ where

- $\delta(q_i, 0) = q_{i+1}$ for $0 \leq i \leq n - 1$
- $\delta(q_i, 1) = q_{i+1}$ for $1 \leq i \leq n - 1$
- $\delta(q_0, 0) = \delta(q_0, 1) = q_0$

Now, we will prove that any DFA D_{C_n} cannot recognize C_n with fewer than 2^n states.

Say a DFA, D_{C_n}, recognizes C_n with $m < 2^n$ states. We take two different strings w_1 and w_2, both of length n. Let q_a and q_b be the states D_{C_n} stops at after reading w_1 and w_2, respectively. Since there are 2^n different strings of length n and $m < 2^n$, by the Pigeonhole Principle, for some w_1-w_2 pair, $q_a = q_b$. Say w_1 and w_2 differs at the j^{th} bit. Then, D_{C_n} reach at the same state when it reads $w_1' = w_1 0^{j-1}$ or $w_2' = w_2 0^{j-1}$, but D_{C_n} should accept one and reject the other. Hence, there is a contradiction, and we can conclude that a DFA needs at least 2^n states to recognize C_n. □

3.2. Simulation of 2DFA by pDFA

Simulation of 2DFA by DFA or pDFA has been analyzed a number of times before, and various upper bounds, which are very close to each other (and bounded by $O(n^n)$), are introduced by different authors [2], [8], [16]. We will first prove the latest upper bound for the conversion of 2DFA to pDFA (i.e. [16]). Then, we will prove the lower bound for this simulation (i.e. [16]) and conclude that the lower bound also matches the upper bound.

Theorem 3.3. Every n-state 2DFA has an equivalent $n(n^n - (n-1)^n)$-state pDFA.

Proof. Kapoutsis answers our questions of concern: The mathematical object to relate 2DFA and pDFA with the least amount of information needed is "tables of 2DFA" (which we will describe below) and one can build the transition function of the simulating pDFA if one can compute a table from another. To formally prove, we will start with the description of tables:

Definition 3.1. Suppose the computation of w on 2DFA D when started at the starting state s on the first symbol of w, shortly denoted by $COMP_{D,s,1}(w)$, hits right into some state p_w. Define the *table of D on w* to be the function

$$TABLE_D(w) = \tau : Q \cup \perp \to Q \tag{3.2}$$

that satisfies $\tau(\perp) = p_w$ and for all $q \in Q$

- $\tau(q) = p$ if $COMP_{D,q,|w|}(w)$ hits right into p,
- $\tau(q) = p_w$ if $COMP_{D,q,|w|}(w)$ hits left, loops, or hangs.

Here are the important points about tables:

- Tables are only defined if $COMP_{D,s,1}(w)$ hits right into some state, meaning that w is promising to be accepted (i.e. does not hang, loop, or hit left). Since

our acceptance condition of a string is, simply, "hitting right into the accepting state", it is thought as unnecessary to keep additional information about the computation of words that cannot hit right.

- One can quickly observe that there can be at most n^{n+1} different tables. The finiteness of this number leads us to the fact that for a chosen 2DFA, two different words can have the same table, analogous to the fact that, for a chosen pDFA, the computation of two different words can stop at the same state.
- Since $\tau(q) = p_w$ for some q, we do not know if this is because $COMP_{D,q,|w|}(w)$ does not hit the rightmost boundary, or because it hits it into p_w, and one can say there occurs a slightly dangerous ambiguity. However, we will go with this definition, and show how it comes useful while introducing Kapoutsis' algorithm for computing tables.
- A table of D on w describes the aforementioned $1 + |Q|$ computations which are regarded as the vital computations. Why do we call these computations $COMP_{D,s,1}(w)$ and $COMP_{D,q,|w|}(w)$ for all $q \in Q$ vital? First of all, the first table entry (i.e. the one about $COMP_{D,s,1}(w)$) represents the small packet of information about the (natural) computation of w and in which state it finally hits right. Secondly, the remaining entries represent another big packet of crucial information: How to compute a table from another one in a relatively easy manner. Say for 2DFA D, the table function τ for some word w is defined and we want to calculate the table function τ' for wa where a is a symbol of our alphabet. Since we know $\tau(\perp)$, we know the state that D hits right on w, and we can find $\tau'(\perp)$ with a short procedure:
  (i) Make temporary state $t = \tau(\perp)$.
  (ii) If $\delta(t, a)$ is not defined (meaning that the computation of wa hangs), then τ' is not defined also. Fail.
  (iii) Else if $\delta(t, a) = (q', r)$ for some $q' \in Q$, then return the result $\tau'(\perp) = q'$.
  (iv) Else if $\delta(t, a) = (q', l)$, then look which state $\tau(q')$ points. Go to the next step.
  (v) If $\tau(q') = \tau(\perp)$ (meaning that the computation of wa goes in a loop, hangs, or hits left because of the behaviour of w), then τ' is not defined. Fail.

(vi) Else if $\tau(q') = t' \neq \tau(\perp)$ and t' is not seen before in the progress of the procedure, then make $t = t'$ and go to step 2. Else, fail.

With a similar procedure, we can also find $\tau'(q)$ for all $q \in Q$:

(i) Make temporary state $t = q$.

(ii) If $\delta(t, a)$ is not defined, then $\tau'(t) = \tau'(\perp)$.

(iii) Else if $\delta(t, a) = (q', r)$ for some $q' \in Q$, then return the result $\tau'(t) = q'$.

(iv) Else if $\delta(t, a) = (q', l)$, then look which state $\tau(q')$ points. Go to the next step.

(v) If $\tau(q') = \tau(\perp)$ (meaning that the computation of wa goes in a loop, hangs, or hits left because of the behaviour of w), then return $\tau'(t) = \tau'(\perp)$.

(vi) Else if $\tau(q') = t' \neq \tau(\perp)$ and t' is not seen before in the progress of the procedure, then make $t = t'$ and go to step 2. Else, $\tau'(t) = \tau'(\perp)$.

Here, we can see the perfect use of this ambiguously-defined table notion, we use it to achieve both our targets: Computing a table from another efficiently and reducing cost of simulation of 2DFA by pDFA in the size of the table function.

- On the empty string, we define ϖ as the constant function in which every entry shows the starting state s since $COMP_{D,s,1}(\epsilon) = s$ (or more generally, since it is defined that $COMP_{D,q,1}(u)$ hits right into q and $COMP_{D,q,|u|}(u)$ hits left into q for $u = \epsilon$).
- On the accepting strings, we define ϱ as the constant function in which every entry shows the final state f because if $TABLE_D(\$u\#)$ is defined on an end-marked u, then $COMP_{D,s,1}(\$u\#) = f$ (i.e. because of the endmarkers violation rules, a natural computation can hit right into f only).

As Kapoutsis states, we can also define tables for 2DFA, not only for 2DFA-word pairs. It is motivated by the following observation: Suppose the table function τ for w is defined and $\tau(\perp) = q$. This means that for some q', $\tau(q') = q$ and $COMP_{D,q,|w|}(w)$ is a suffix of $COMP_{D,s,1}(w)$. This gives a useful insight: $\tau(\perp) \in \tau[Q]$, that is $\tau(\perp)$ is equal to a value assigned by τ for some state element of Q (we will use this property later in the lower bound proof for simulation of 2DFA by DFA). Thus, we will define tables of 2DFA as following:

Definition 3.2. *Table T of D* is any $\tau : Q \cup \perp \to Q$ such that $\tau(\perp) \in \tau[Q]$.

One should remember that tables of 2DFA and tables of 2DFA on some word w are slightly different concepts and we have given definitions for both of them so far. Such characterization of tables of 2DFA gives us the following:

Lemma 3.4. The number of distinct tables of D is $n^{n+1} - n(n-1)^n = n(n^n - (n-1)^n)$.

Proof. The number of tables of D is actually the number of all possible tables excluding the ones where $\tau(\perp) \notin \tau[Q]$. Henceforth, the number of distinct tables of D is equal to $n^{n+1} - n(n-1)^n = n(n^n - (n-1)^n)$. □

Now, we will build a pDFA from a 2DFA D by using the facts and results we gained about tables. We claim that $w = w_1 w_2 ... w_m$ (w_i for $1 \leq i \leq m$ are letters) is accepted by D if and only if there is a sequence of tables $\mathcal{T}$ of D where $\mathcal{T}$ is:

$$TABLE_D(\epsilon), TABLE_D(\$), TABLE_D(\$w_1), TABLE_D(\$w_1w_2), ..., TABLE_D(\$w\#) \tag{3.3}$$

in which the $i+1^{th}$ table $TABLE_D(ua)$'s function is equal to the table that can be computed from the previous i^{th} table $TABLE_D(u)$ by using the procedure described above (u is a prefix of $\$w\#$ and a is a letter). One can see that both directions of the claim holds: If there really exists such a sequence, then w should be accepted (because of the points about tables we discussed) and trivially, such an acceptance directly says that a sequence with characteristics above exists. Based on this result, we can use tables of 2DFA as the states and the constant table functions ϖ and ϱ are the starting and accepting states of our pDFA that simulates 2DFA, respectively. The transition function of pDFA is also simple: After reading the prefix u of the word ua, on reading a symbol a and being on state $q = TABLE_D(u)$, the pDFA moves into the state $q' = TABLE_D(ua)$, and if such q' does not exist, it halts and rejects. Hence, we can conclude that every 2DFA with n states has an equivalent pDFA with at most $n(n^n - (n-1)^n)$ states. □

Theorem 3.5. Some n-state 2DFA has an equivalent $n(n^n - (n-1)^n)$-state minimal pDFA.

Proof. To prove this theorem, we will first define a language and then show why and how the smallest pDFA needs $n(n^n-(n-1)^n)$ states to decide that language. To define our language of concern, first we should describe the concept of promise problems which were brought into the literature by Even, Selman, and Yacobi [23].

A "promise problem" is a partition of the set of all strings into three subsets: "YES-strings", "NO-strings", and "ignored strings". Basically, the automaton solving a given promise problem accepts YES-strings and rejects NO-strings; we do not care if this machine accepts or rejects the ignored strings because we promise that such strings are not of concern. When there are no ignored strings, the problem is called language as we are more familiar with.

Here, the alphabet of the promise problem we consider is $\Gamma = ([n]+\mathcal{P}([n]\times[n]))\times\{l,r\}$, where $[n]$ represents the integer interval $\{1,2,...,n\}$ and $\{l,r\}$ are the direction tags for left and right. Specifically, we are interested in the 4-character-pair-long words of the following form:

$$w = (x,l)(g,l)(h,r)(y,r) \tag{3.4}$$

where x and y are numbers in $[n]$ and g and h are partial functions in $\mathcal{P}([n]\times[n])$.

It is beneficial for the readers to visualize these special words and their acceptance conditions as directed bipartite graphs and paths between selected nodes. The critical part here is to see (g,l) and (h,r) as representations of sets of arrows between nodes of n-node graphs. That in mind, the promise problem $\Psi = (\Psi_{yes}, \Psi_{no})$ with

$$\Psi_{yes} = \{w \in \Gamma^* \mid w \text{ describes a path}\},\ \Psi_{no} = \{w \in \Gamma^* \mid w \text{ does not describe a path}\} \tag{3.5}$$

can be solved by a 2DFA M with n states. As Kapoutsis states in [16], the operating procedure of M on input $w = (x, l)(g, l)(h, r)(y, r)$ is relatively straightforward:

> Starting from the left endmarker, first reach to (g, l) at state x. Then, alternately read (g, l) and (h, r) and repeatedly follow the arrows (if any) defined by g and h. If we reach (h, r) at a state y' from which no h-arrow departs, we stay at y' and move right to check whether $y' = y$. If so, get past the left endmarker and accept. In all other cases, reject.

Formally, $M = (\{1, 2, ..., n\}, \Gamma_{\$,\#}, \delta, 1, 1)$ and δ is any partial function $[n] \times \Gamma_{\$,\#} \to [n] \times \{l, r\}$ satisfying the following equations:

- $\delta(1, \$) = (1, r), \quad \delta(1, (x, l)) = (x, r), \quad \delta(1, \#) = (1, r),$
- $\delta(z, (g, l)) = (g(z), r)$ if $g(z)$ is defined; otherwise $\delta(z, (g, l))$ is undefined,
- $\delta(z, (h, r)) = (h(z), l)$ if $h(z)$ is defined; otherwise $\delta(z, (h, r)) = (z, r)$,
- $\delta(z, (y, r)) = (1, r)$ if $z = y$; otherwise $\delta(z, (y, r))$ is undefined.

One quick note is that if there is a path from x to y, then there should not be an h-arrow departing from y. This means that if one can go from x to y by using g-arrows and h-arrows and there is an h-arrow departing from y, then we say there is not a path from x to y. We define paths in this slightly unnatural way.

To be precise about our lower bound proof, we now give a special form of Ψ and show that the smallest DFA needs $n(n^n - (n-1)^n)$ states to solve it. We state that the promise problem $\Psi' = (\Psi'_{yes}, \Psi'_{no})$ with

- $\Psi'_{yes} = \{w_{\tau,\tau'} \mid \tau, \tau' \text{ are tables of } M \text{ and } w_{\tau,\tau'} \text{ has a path}\}$
- $\Psi'_{no} = \{w_{\tau,\tau'} \mid \tau, \tau' \text{ are tables of } M \text{ and } w_{\tau,\tau'} \text{ has no path}\}$

is solved by our n-state 2DFA M where $w_{\tau,\tau'} = (x_\tau, l)(g_\tau, l)(h_{\tau,\tau'}, r)(y_{\tau,\tau'}, r)$ and $x_\tau = \min\{x \mid \tau(x) = \tau(\perp)\}$ and $g_\tau = \{(x, \tau(x)) \mid x \in [n]\}$. Before the end of this proof, we will surely define $y_{\tau,\tau'}$ and $h_{\tau,\tau'}$ explicitly, but first we explain why to bring such a complicated word into the table.

Now, let us take a table τ of M and restrict its domain to $[n]$. See that a partial function g can be used to describe such a restricted table. Now, we claim that there is a way to distinguish every distinct table of M. The trick is to show that given tables τ and τ', introducing a cleverly defined reverse-arrow in both tables (i.e. arrows from the range to the domain of these tables) results in creating a path in only one of the tables. Briefly, we mean that we can differentiate every given pair of tables of M by computing on special words which are used to represent those tables and we do that by using M. We do this reverse-arrow manipulation part especially with the help of h and y. There are three key points to understand and digest this idea:

(i) We want M to differentiate its own tables by doing computation on the words that more or less represent its tables.

(ii) The prefix $\$(x_\tau, l)(g_\tau, l)$ is designed so that the table of M on this prefix is exactly τ.

(iii) As Kapoutsis proves, two tables τ and τ' of M are distinct if and only if there exist a partial function $h : [n] \to [n]$ and a $y \in [n]$ such that exactly one of the following inputs has a path: $(x_\tau, l)(g_\tau, l)(h, r)(y, r)$ or $(x_{\tau'}, l)(g_{\tau'}, l)(h, r)(y, r)$. Suppose these tables are distinct. To guarentee only one path, we can choose h to be the empty function and y to be the smallest among $\tau(\perp)$ and $\tau'(\perp)$ if $\tau(\perp) \neq \tau'(\perp)$. If $\tau(\perp) = \tau'(\perp) = z$, we can choose h to contain only one arrow from z to the smallest x where $\tau(x) \neq \tau'(x)$, and y to be the smallest of $\tau(x)$ and $\tau'(x)$ that is different from z. This critical argument allows us to distinguish one table from another (by the way, the other direction of this if and only if statement is trivial; if two tables are not distinct, then they are identical and there cannot be such h and y to distinguish them).

By the way, we want to clear something. When we say "we choose the smallest (or minimum) among a set of states" we mean that those states are numerically sorted and we choose the numerically smallest one. For example q_1 is smaller than q_3 since $1 < 3$, for a given set of states $\{q_1, q_2, q_3, q_4\}$.

Now is the time to set the remaining variables $y_{\tau,\tau'}$ and $h_{\tau,\tau'}$: If $\tau \neq \tau'$, $h_{\tau,\tau'}$ and $y_{\tau,\tau'}$ take the values given in the third point; else if $\tau = \tau'$, $y_{\tau,\tau'} = \tau(x_\tau)$ and $h_{\tau,\tau'} = \emptyset$. Before going further, one might want to note that the structure of our special words guarantee us to restate a basic fact that every table has a path to itself since

$$w_{\tau,\tau} = (x_\tau, l)(g_\tau, l)(h_{\tau,\tau}, r)(y_{\tau,\tau}, r) \tag{3.6}$$

has a very simple path with $h_{\tau,\tau} = \emptyset$ and $y_{\tau,\tau} = \tau(x_\tau) = \tau(\perp)$. Now, we will prove that solving Ψ' costs at least $n(n^n - (n-1)^n)$ states for some pDFA.

Suppose a pDFA M' solves our promise problem with $m' < n(n^n-(n-1)^n)$ states. For every table τ of M, $w_{\tau,\tau}$ is accepted by M and M'. That means there is a state q_τ of M' that reading the prefix $\$(x_\tau, l)(g_\tau, l)$ results in M' to go into q_τ, and since there are $m' < n(n^n - (n-1)^n)$ states, there happen to be a table $\tau' \neq \tau$ and a word $w_{\tau',\tau'}$ such that M' goes into the same state q_τ after reading $\$(x_{\tau'}, l)(g_{\tau'}, l)$. Furthermore, starting from q_τ, M' should always reach into an accepting state after reading suffices $(h_{\tau,\tau}, r)(y_{\tau,\tau}, r)\#$ and $(h_{\tau',\tau'}, r)(y_{\tau',\tau'}, r)\#$, considering they are the suffices of $w_{\tau,\tau}$ and $w_{\tau',\tau'}$, respectively. But that means M' does the very same computation on both $w_{\tau',\tau}$ and $w_{\tau,\tau'}$ too, and accepts them both, resulting in a contradiction to the third point we discuss above in this proof. Thus, M' with m' states is not adequate to solve Ψ' and our main argument is correct: there exists an n-state 2DFA that has an equivalent $n(n^n - (n-1)^n)$-state minimal pDFA. □

We do not know whether there exists a regular language with a much smaller alphabet size that can be decided by an n-state 2DFA and a $n(n^n - (n-1)^n)$-state minimal pDFA. Thus, it is possible that the largeness of the alphabet size of Ψ', or using a promise problem instead of a regular language can be necessary for the lower bound.

3.3. Simulation of 2NFA by pDFA

Simulation of 2NFA by DFA or pDFA has been analyzed a number of times before, and various upper bounds, which are very close to each other (and bounded by $O(2^{n^2})$), are introduced by different authors [2], [8], [16]. We will first prove the latest upper bound for the conversion of 2NFA to pDFA (i.e. [16]). Then, we will prove the lower bound for this simulation (i.e. [16]) and conclude that the lower bound also matches the upper bound.

Theorem 3.6. Every n-state 2NFA N has an equivalent $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state pDFA.

Proof. Kapoutsis answers our questions of concern: The mathematical object to relate 2NFA and pDFA with the least amount of information needed is "tables of 2NFA" (which we will describe below) and one can build the transition function of the simulating pDFA if one can compute a table from another. Similar to the 2DFA-pDFA case, we will first define tables of 2NFA, then we will give the number of distinct tables of 2NFA, and finally we will show a way to compute one table from another. We start with the description of tables:

Definition 3.3. Suppose the set of computations of w on 2NFA N when started at the starting state s on the first symbol of w, shortly $COMP_{N,s,1}(w)$, hits right into some non-empty set of states, denoted by P_w. Define the *table of N on w* to be the function

$$TABLE_N(w) = T : Q \cup \bot \to \mathcal{P}(Q) \setminus \emptyset \tag{3.7}$$

that satisfies $T(\bot) = P_w$ and for all $q \in Q$

- $T(q) = P \setminus P_w$ if $COMP_{N,q,|w|}(w)$ hits right into some $P \nsubseteq P_w$,
- $T(q) = P_w$ if $COMP_{N,q,|w|}(w)$ hits right into some $P \subseteq P_w$.

Here are some important points about tables:

- The notion of "tables of a 2NFA on a word" is meant to be a generalization of "tables of a 2DFA on a word". Thus, it produces a similar ambiguity: whenever $T(q) = P_w$ we do not know if this is because all computations in $COMP_{N,s,|w|}(w)$ miss the right boundary (i.e. $P = \emptyset$) or because some of them hit it but do so only into states that are already in P_w. Again, Kapoutsis turns this ambiguity into advantage in simulation of 2NFA by pDFA.
- We will show to compute a table from another one in a relatively easy manner. Say for 2NFA N, the table function T for w is defined and we want to calculate the table function T' for wa where a is a symbol of our alphabet. Since we know $T(\perp)$, we know the state set that N hits right on w, and we can find $T'(\perp)$ with a procedure that is actually the generalization of the table finding algorithm given for 2DFA-pDFA simulation case:
  (i) Make the temporary set of states $P = T(\perp)$. Make the temporary promising set of states $S' = \emptyset$.
  (ii) Make set $R = \bigcup\{\delta(p, a) \mid p \in P\}$ (to investigate $\delta(p, a)$ wholly).
  (iii) Make set $S' = S' \cup \{q \mid (q, r) \in R\}$ (to distinguish the computations that go right).
  (iv) Make set $R' = \bigcup\{T(q) \mid (q, l) \in R\}$ (to find the table values on the computations that go left).
  (v) Update P as $P = \{p \in R' \mid (p, r) \notin R \text{ or } (p, l) \notin R\}$. If $P = \emptyset$, go to the next step; otherwise go to the second step.
  (vi) If $S' = \emptyset$, fail; otherwise return S'.

  With a similar procedure, we can also find $T'(q)$ for all $q \in Q$:
  (i) Make the temporary set of states $P = \{q\}$. Make the temporary promising set of states $S' = \emptyset$.
  (ii) Make set $R = \bigcup\{\delta(p, a) \mid p \in P\}$ (to investigate $\delta(p, a)$ wholly).
  (iii) Make set $S' = S' \cup \{q \mid (q, r) \in R\}$ (to distinguish the computations that go right).
  (iv) Make set $R' = \bigcup\{T(q) \mid (q, l) \in R\} \setminus T(\perp)$. (to find the table values on the computations that go left while obeying the rules about tables).

(v) Update P as $P = \{p \in R' \mid (p, r) \notin R \text{ or } (p, l) \notin R\}$. If $P = \emptyset$, go to the next step; otherwise go to the second step.

(vi) If $S' \subseteq T'(\perp)$, return $T'(\perp)$; otherwise return $S' \setminus T'(\perp)$ (obeying the rules about tables).

- On the empty string, we define ϖ as the constant function in which every entry shows the set of the starting state $\{s\}$ since $COMP_{N,s,1}(\epsilon) = \{s\}$ (or more generally, since it is defined that $COMP_{N,q,1}(u)$ hits right into $\{q\}$ and $COMP_{N,q,|u|}(u)$ hits left into $\{q\}$ for $u = \epsilon$).
- On the accepting strings, we define ϱ as the constant function in which every entry shows the set of the final state $\{f\}$ because if $TABLE_N(\$u\#)$ is defined on an end-marked u, then $COMP_{N,s,1}(\$u\#) = \{f\}$ (i.e. because of the endmarkers violation rules, a natural computation can hit right into $\{f\}$ only).

As one can see, on some word w, table definition of 2NFA are slightly different from that of 2DFA. We will show that tables of 2NFA are also different than tables of 2DFA, so we want to recall what Kapoutsis says about tables of 2NFA:

For a 2NFA, there may exist more than one computation for a word w, so we have the set of all computations C in the 2NFA case rather than just a computation c in the 2DFA case. Suppose the table $T = TABLE_N(w)$ is defined. Now suppose $COMP_{N,s,1}(w)$ hits right into the set of states $P_w = T(\perp) \neq \emptyset$. Take one of them, say c. c visits the rightmost symbol of w at least once. If p is the state of N at one of these visits, then combining the prefix of c up to that visit with any of the right-hitting computations in $COMP_{N,p,|w|}(w)$, gives a right-hitting computation which is also in C. Therefore, the computations of $COMP_{N,p,|w|}(w)$ can hit right only into states that are already in $T(\perp)$. The definition of T with the previous argument implies that $T(p) = T(\perp)$. Furthermore, the definition of T reveals that every state that is not assigned to the set $T(\perp)$ is assigned a set disjoint from $T(\perp)$; otherwise it has to be equal to $T(\perp)$. Thus, we will define tables of 2NFA as following:

Definition 3.4. A *table of* N is any $T : Q \cup \perp \to \mathcal{P}(Q) \setminus \emptyset$ such that

- for every $p \in Q$, $T(p) = T(\perp)$ or $T(p) \cap T(\perp) \neq \emptyset$,
- for some $p \in Q$, $T(p) = T(\perp)$.

One should remember that tables of 2NFA and tables of 2NFA on some word w are slightly different concepts and we have given definitions for both of them so far. Such characterization of tables of 2NFA gives us the following:

Lemma 3.7. The number of distinct tables of N is $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$.

Proof. The number of distinct tables of N is equal to the number of distinct $(n+1)$-tuples of non-empty subsets of the set $\{1, 2, ..., n\}$ where the set in the first component (i.e. $T(\perp)$) intersects no other set in the tuple but can appear in the other components. For each $i, j = 1, 2, ..., n$, there are $\binom{n}{i}$ choices for the first component (i.e. the number of all possible cases that i different states are chosen among n states) and $\binom{n}{j}$ choices for the set of the components after it that has the same set (i.e. which j components that are equal to the first one are chosen). Given i and j, each one of the remaining $(n+1)-(j+1) = n-j$ components can take any of the $2^{n-i}-1$ non-empty sets that avoid intersection with the first component. Overall, we have

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\binom{n}{i}\binom{n}{j}(2^{n-i}-1)^{n-j} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j \tag{3.8}$$

choices for this $(n+1)$ tuple. Right of this equation comes by taking $i = n-i$ and $j = n-j$ and reaarranging the sum accordingly. □

Now, we will build a pDFA from a 2NFA N by using the facts and results we gained about tables and this proof is very similar to the one shown in 2DFA-pDFA case. We claim that $w = w_1w_2...w_m$ (w_i for $1 \leq i \leq m$ are letters) is accepted by N if

and only if there is a sequence of tables $\mathcal{T}$ of N where $\mathcal{T}$ is:

$$TABLE_N(\epsilon), TABLE_N(\$), TABLE_N(\$w_1), TABLE_N(\$w_1w_2), ..., TABLE_N(\$w\#) \tag{3.9}$$

in which the $i+1^{th}$ table $TABLE_N(ua)$'s function is equal to the table that can be computed from the previous i^{th} table $TABLE_N(u)$ by using the procedure described above (u is a prefix of $\$w\#$ and a is a letter). One can see that both directions of the claim holds: If there really exists such a sequence, then w should be accepted (because of the points about tables we discussed) and trivially, such an acceptance directly says that a sequence with characteristics above exists. Based on this result, we can use these tables as the states and the constant table functions ϖ and ϱ are the starting and accepting states of our pDFA that simulates 2NFA, respectively. The transition function of pDFA is also simple: After reading the prefix u of the word ua, on reading a symbol a and being on state $q = TABLE_N(u)$, the pDFA moves into the state $q' = TABLE_N(ua)$, and if such q' does not exist, it halts and rejects. Hence, we can conclude that every 2NFA with n states has an equivalent $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state pDFA. □

Theorem 3.8. Some n-state 2NFA has an equivalent $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state minimal pDFA.

Proof. To prove this theorem, we will first define a promise problem and then show why and how the smallest pDFA needs $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$ states to decide that.

In this proof, the alphabet of the promise problem we consider is $\Gamma = ([n] + \mathcal{P}([n] \times [n])) \times \{l, r\}$, where $[n]$ represents the integer interval $\{1, 2, ..., n\}$ and $\{l, r\}$ are the direction tags for left and right. Specifically, we are interested in the 4-character-pair-long words of the following form:

$$w = (x, l)(G, l)(h, r)(y, r) \tag{3.10}$$

where x and y are numbers in $[n]$, G is a binary relation on $[n]$, and h is a partial function.

As one can see, the words we use here are similar to the ones described in 3.5, the lower bound proof of 2DFA-pDFA case. The difference is that there is a binary relation here, not a partial function, as defined in the 2DFA-pDFA case. One may want to visualize the difference between binary relations and partial functions as the following: A set of n nodes all having 0 or more outgoing arrows to a different set of n nodes can represent binary relations whereas a set of n nodes all having at most 1 outgoing arrow to a different set of n nodes can represent partial functions.

As is in the 2DFA-pDFA lower bound case, it is beneficial for the readers to visualize these special words and their acceptance conditions as directed bipartite graphs and paths between selected nodes. The critical part here is to see (G, l) and (h, r) as representations of sets of arrows between the nodes of n-node graphs. That in mind, the promise problem $\Pi = (\Pi_{yes}, \Pi_{no})$ with

$$\Pi_{yes} = \{w \in \Gamma^* \mid w \text{ describes a path}\},\ \Pi_{no} = \{w \in \Gamma^* \mid w \text{ does not describe a path}\}$$

can be solved by a 2NFA M with n states. As Kapoutsis states in [16], the operating procedure of M on input $w = (x, l)(G, l)(h, r)(y, r)$ is relatively straightforward:

> Starting from the left endmarker, first reach to (G, l) at state x. Then, alternately read (G, l) and (h, r) and using nondeterminism check all possible paths by repeatedly following the arrows (if any) defined by G and h. If by any one of nondeterministic branches, we reach (h, r) at a state y' from which no h-arrow departs, we stay at y' and move right to check whether $y' = y$. If so, get past the left endmarker and accept. In all other cases, reject.

Formally, $M = (\{1, 2, ..., n\}, \Gamma_{\$,\#}, \delta, 1, 1)$ and δ is any total function $[n] \times \Gamma_{\$,\#} \to \mathbb{P}([n] \times \{l, r\})$ satisfying the following equations:

- $\delta(1, \$) = \{(1, r)\}$, $\delta(1, (x, l)) = \{(x, r)\}$, $\delta(1, \#) = \{(1, r)\}$,
- $\delta(z, (G, l)) = \{(z', r) | (z, z') \in G\}$,

- $\delta(z,(h,r)) = \{(h(z),l)\}$ if $h(z)$ is defined; otherwise $\delta(z,(h,r)) = \{(z,r)\}$,
- $\delta(z,(y,r)) = \{(1,r)\}$ if $z = y$; otherwise $\delta(z,(y,r)) = \emptyset$.

As in the 2DFA-pDFA case 3.5, if there is a path from x to y, then there should not be an h-arrow departing from y. This means that if one can go from x to y by using g-arrows and h-arrows and there is an h-arrow departing from y, then we say there is not a path from x to y. Again, we define paths in this slightly unnatural way.

To be precise about our lower bound proof, we now give a special form of Π and show that the smallest DFA needs $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$ states to solve it. We state that the promise problem $\Pi' = (\Pi'_{yes}, \Pi'_{no})$ with

- $\Pi'_{yes} = \{w_{T,T'} \mid T, T' \text{ are tables of } M \text{ and } w_{T,T'} \text{ has a path}\}$
- $\Pi'_{no} = \{w_{T,T'} \mid T, T' \text{ are tables of } M \text{ and } w_{T,T'} \text{ has no path}\}$

is solved by our n-state 2NFA M where $w_{T,T'} = (x_T,l)(G_T,l)(h_{T,T'},r)(y_{T,T'},r)$ and $x_T = \min\{x \mid T(x) = T(\perp)\}$ and $G_T = \{(x,y) \mid y \in T(x)\}$. A little later, we will define $y_{T,T'}$ and $h_{T,T'}$ explicitly.

For some 2NFA M, let us take a table T of M and restrict its domain to $[n]$. See that a binary relation G can be used to describe such a restricted table. Now, we claim that there is a way to distinguish every distinct table of M. The trick is to show that given tables T and T', introducing a cleverly defined reverse-arrow in both tables (i.e. arrows from the range to the domain of these tables) results in creating a path in only one of the tables. Briefly, we mean that we can differentiate every given pair of tables of M by computing on some special words which are used to represent those tables and we do that by using M. We do this reverse-arrow manipulation part especially with the help of h and y. There are three key points to understand this idea:

(i) We want M to differentiate its own tables by doing computation on the words that more or less represent its tables.

(ii) The prefix $\$(x_T, l)(G_T, l)$ is designed so that the table of M on this prefix is exactly T.

(iii) As Kapoutsis proves, two tables T and T' of M are distinct if and only if there exist a partial function $h : [n] \to [n]$ and a $y \in [n]$ such that exactly one of the following inputs has a path: $(x_T, l)(G_T, l)(h, r)(y, r)$ or $(x_{T'}, l)(G_{T'}, l)(h, r)(y, r)$. Suppose these tables are distinct. To guarentee only one path, we can choose h to be the empty function and y to be the smallest state in the symmetric difference of $T(\perp)$ and $T'(\perp)$ if $T(\perp) \neq T'(\perp)$. If $T(\perp) = T'(\perp) = Z$, we can choose h to contain the arrows from Z to the smallest x where $T(x) \neq T'(x)$, namely $h = \{(z, x) | z \in Z\}$,, and y to be the smallest element in the symmetric differrence of $T(x)$ and $T'(x)$. This critical argument allows us to distinguish one table from another (by the way, the other direction of this if and only if statement is trivial; if two tables are not distinct, then they are identical and there cannot be such h and y to distinguish them).

Now is the time to set the remaining variables $y_{T,T'}$ and $h_{T,T'}$: If $T \neq T'$, $h_{T,T'}$ and $y_{T,T'}$ take the values given in the third point; else if $T = T'$, $y_{T,T'} = \min T(x_T)$ and $h_{T,T'} = \emptyset$. Before going further, one might want to note that the structure of our special words guarantee us to restate a basic fact that every table has a path to itself since

$$w_{T,T} = (x_T, l)(G_T, l)(h_{T,T}, r)(y_{T,T}, r) \tag{3.11}$$

has a very simple path with $h_{T,T} = \emptyset$ and $y_{T,T} = T(x_T) = T(\perp)$. Now, we will prove that solving Π' costs at least $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \binom{n}{i} \binom{n}{j} (2^i - 1)^j$ states for some pDFA.

Suppose a pDFA M' solves our problem with $m' < \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \binom{n}{i} \binom{n}{j} (2^i - 1)^j$ states. For every table T of M, $w_{T,T}$ is accepted by M and M'. That means there is a state q_T of M' that reading the prefix $\$(x_T, l)(G_T, l)$ results in M' to go into q_T, and since there are $m' < \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \binom{n}{i} \binom{n}{j} (2^i - 1)^j$ states, there happen to be

a table $T' \neq T$ and a word $w_{T',T'}$ such that M' goes into the same state q_T after reading $\$(x_{T'}, l)(G_{T'}, l)$. Furthermore, starting from q_T, M' should always reach into an accepting state after reading suffices $(h_{T,T}, r)(y_{T,T}, r)\#$ and $(h_{T',T'}, r)(y_{T',T'}, r)\#$, considering they are the suffices of $w_{T,T}$ and $w_{T',T'}$, respectively. But that means M' can produce two computations on both $w_{T',T}$ and $w_{T,T'}$ too, which are exactly the same, and accepts them both, resulting in a contradiction to the third point we discuss above in this proof. Thus, M' with m' states is not adequate to solve Π' and our main argument is correct: there exists an n-state 2NFA that has an equivalent $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state minimal pDFA. □

As in the 2DFA-pDFA case, we do not know whether there exists a regular language with a much smaller alphabet size that can be decided by an n-state 2NFA and a $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state minimal pDFA. Thus, it is possible that the largeness of the alphabet size of Ψ', or using a promise problem instead of a regular language can be necessary for the lower bound.

3.4. Simulation of AFA by DFA

In this section, we will prove some important theorems about AFA and its relation to DFA. We start with the following:

Theorem 3.9. Every AFA with n states has an equivalent DFA with at most 2^{2^n} states.

Proof. The correct mathematical object to relate the states of AFA to those of DFA is, basically, the "power set of power set". More precisely, given an AFA $D = (Q, \Sigma, \delta, q_0, F)$, the all-CDNF set $CDNF_Q$ of Q, which has the size $2^{2^{|Q|}}$, can be used as the set of states of DFA that is equivalent to D.

A clause in a member of $CDNF_Q$ is of size $n = |Q|$ since it has exactly n variables in conjunction. A basic counting argument leads us that there can be $m = 2^n$ different

clauses. Furthermore, since a CDNF is just a disjunction of at most 2^m clauses (e.g. power set of 2^Q), there can be at most $M = 2^m = 2^{2^n}$ different CDNFs. Thus, M states are enough for a DFA to simulate such AFA. Now, we will describe how to define the transition function of such DFA.

Since a CDNF is a state of DFA and we know the way AFA compute, which is basically building a CDNF (i.e. the next propositional formula) from the current CDNF (i.e. the current propositional formula) given a symbol a, it is easy to define the DFA's transitions: For every possible CDNF-state, give a to it and calculate the resulting CDNF according the transition function of the given AFA. Here, we give the CDNF to CDNF calculation algorithm:

Suppose we have a CDNF formula for the word w, denoted as $f(w)$ and we want to compute the CDNF formula $f(w')$ for the word $w' = wa$, where $a \in \Sigma$. For every state q in $f(w)$, we replace it with the CDNF $\delta(q, a)$. Now, we have the resulting more complex formula $f'(w')$. Actually, this complex formula can be seen as a CDNF of CDNFs rather than our usual CDNF of variables, meaning that the clauses inside this complex formula consist of CDNFs instead of variables. Then, by using logical equivalences, we shorten $f'(w')$ into the CDNF-state $f(w')$. Actually, what we do here is using the extended transition function definition 2.29. Now, we analyze space and time requirements of this shortening procedure.

In a CDNF, every clause has n variables and there are at most 2^n clauses, hence a CDNF has at most $n2^n$ variables. In a complex CDNF as we described above (i.e. CDNF of CDNFs), every clause has a CDNF, so every clause has $n(n2^n) = n^2 2^n$ variables; henceforth a complex CDNF has $2^n(n^2 2^n) = n^2 2^{2n}$ variables. Such a shortening procedure for a boolean formula that has $n^2 2^{2n} = O(2^n)$ variables surely is timewise costly and runs in $O(2^{2^n})$ (i.e. given a boolean function with n variables, it takes $O(2^n)$ operations using logical equivalences to convert into CDNF), but considering the fact that we need to generate and store 2^{2^n} states for our equivalent DFA, this shortening procedure is timewise comparable to state generation procedure: Both takes $O(2^{2^n})$ running time. Hence, we can build a DFA that is equivalent to some given AFA. $\square$

2^{2^n} is the upper bound for the simulation of AFA by DFA. This simulation upper bound matches with the ones given in [4], [17], [6], and [7] for other variations of AFA. Later, we will prove that the lower bound matches this upper bound, but first, we need to prove another theorem that is very relevant.

Theorem 3.10. Say some language L is decided by a DFA D_L with n states. Then, the reverse language of L, namely L^R, is decided by an AFA A_{L^R} with $\lceil \log n \rceil$ states [6].

Proof. Deciding the reverse of a language is an interesting procedure. Now, we will analyze this.

Suppose a DFA $D_L = (Q, \Sigma, \delta, q_0, F)$ is given, deciding some language L. For a given non-empty word s, s can be described as concatenation of other words $u, w \in \Sigma^*$ and a symbol v, say $s = uvw$. Suppose D_L starts to read s and is in some state p after reading u. On $\delta(p, v) = q$, it jumps into the state q, and then continues to read the rest of s, namely w. If it finishes reading s on some accepting state, say $f \in F$, it accepts s.

Now suppose we want to decide the reverse of L, namely L^R, and so we would like to accept s^R, the reverse of s, which can be written as $s^R = w^R v u^R$. If we could modify δ in some way to revert the computation mechanism, it would be very beneficial for us to construct a machine to decide L^R. Specifically, with the guidance gained from the information about D_L and how it reads s, we would like to build a machine such that on s^R:

(i) starts in the state f,
(ii) is in the state q after reading w^R,
(iii) jumps from the state q into the state p after reading v, and
(iv) ends in q_0, which is also one of its accepting states, after reading the rest of s^R, namely u^R.

Even though we will not build an AFA that literally has all these four features, we will make use of them while constructing A_{L^R}. Now, we turn to AFA.

For some fixed symbol $a \in \Sigma$, we can visualize every transition of an DFA as a gate that takes 1 input and produce 1 output (e.g. $\delta(q, a) = p$, p as input and q as output). Now suppose that we write every state of D_L in binary form. That way, we can further say that the transitions can be seen as gates that take $\lceil \log n \rceil$ bits as input and produce $\lceil \log n \rceil$ bits as output. One way to represent those input bits is to use binary variables and their complements (a similarity to the states in AFA) where 1 is assigned for a variable x_i and 0 is assigned for its complement $\bar{x}_i$. A table describing this representation shift is shown in table 3.1 for an example DFA $D_L = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, q_0, \{q_0\})$ (δ is given in the table):

Table 3.1. AFA-DFA representation shift, 1.

Representation shift on some fixed symbol $a \in \Sigma$					
states	$\delta(q, a)$	input	output	input with variables	output
q_0	q_1	00	01	$\bar{x}_1\ \bar{x}_0$	01
q_1	q_2	01	10	$\bar{x}_1\ x_0$	10
q_2	q_3	10	11	$x_1\ \bar{x}_0$	11
q_3	q_1	11	01	$x_1\ x_0$	01

Furthermore, we can represent output bits by using boolean functions y_is. Now, we will use the gate notion effectively in the sense of logic gates. We can think of the transition function δ as a combination of logic gates that do "AND" and "OR" operations on $\lceil \log n \rceil$ variables x_is and $\bar{x}_i$s, and return $\lceil \log n \rceil$ functions producing output in the form of y_is. Mirroring our argument, we can perceive the fifth and sixth columns of the previous table as a truth table for x_is and y_is. The continued representation shift is shown in table 3.2:

Table 3.2. AFA-DFA representation shift, 2.

Representation shift on some fixed symbol $a \in \Sigma$ (continued)				
input variables	output	input x_is	y_1	y_0
$\bar{x}_1\ \bar{x}_0$	01	$\bar{x}_1 \wedge \bar{x}_0$	0	1
$\bar{x}_1\ x_0$	10	$\bar{x}_1 \wedge x_0$	1	0
$x_1\ \bar{x}_0$	11	$x_1 \wedge \bar{x}_0$	1	1
$x_1\ x_0$	01	$x_1 \wedge x_0$	0	1

Furthermore, we can see every y_i as boolean functions that gives 1 or 0 on specific conjunctions of x_is and $\bar{x}_i$s. To clearly define y_is, we can use CDNFs of such x_is and $\bar{x}_i$s that gives 1 (i.e. "true"). Finally, here comes the part about reverting: If we can represent the computation on the word s as rules about doing binary operations on input variables x_is producing outputs y_is, we can represent the computation on s^R as the reverse of these rules. For example, on a symbol a, $\delta(((\bar{x}_1 \wedge x_0) \vee (x_1 \wedge \bar{x}_0)), a) = y_0$ becomes $\delta'(z_0, a) = (\bar{z}_1 \wedge z_0) \vee (z_1 \wedge \bar{z}_0)$. The final representation shift is shown in table 3.3:

Table 3.3. AFA-DFA representation shift, 3.

Representation reversion on some fixed symbol $a \in \Sigma$						
variables x_is	y_1	y_0	formula	output	states	δ'
$\bar{x}_1 \wedge \bar{x}_0$	0	1	$(\bar{x}_1 \wedge x_0) \vee (x_1 \wedge \bar{x}_0)$	y_0	z_0	$(\bar{z}_1 \wedge z_0) \vee (z_1 \wedge \bar{z}_0)$
$\bar{x}_1 \wedge x_0$	1	0	$(\bar{x}_1 \wedge \bar{x}_0) \vee (x_1 \wedge \bar{x}_0) \vee (x_1 \wedge x_0)$	y_1	z_1	$(\bar{z}_1 \wedge \bar{z}_0) \vee (z_1 \wedge \bar{z}_0) \vee (z_1 \wedge z_0)$
$x_1 \wedge \bar{x}_0$	1	1				
$x_1 \wedge x_0$	0	1				

Here, z_0 and z_1 mirror y_0 and y_1 as inputs, and z_0 and z_1 mirror x_0 and x_1 as outputs. As one can see, the sixth and seventh columns of the previous table together represent the transition function of some AFA. Thus, we almost-successfully describe a

way to change a DFA deciding some language into an AFA deciding the reverse of that language. We say "almost", because we have not given the starting formula and the set of final states of this AFA. At the beginning of this proof, we have given an abstract computation reversion mechanism and stated that starting state and final states of DFA deciding some L are more or less the final states and the starting formula of AFA deciding L^R, respectively. Formally, the starting formula of this AFA should be the final states of DFA that are written in binary variable form (disjunction of conjunctions), and the final state set of this AFA should be consisting of a simple conjunctive formula that represents the starting state of DFA. For example, the starting formula q_0 of D_L now defines the final state set of the AFA deciding L_R, say A_{L^R}, and it is the empty set $\emptyset$ (i.e. $q_0 \implies$ "$x_0 = 0$ and $x_1 = 0$" $\implies$ "z_0 and z_1 are not final states"). Furthermore, the final state set $\{q_0\}$ of D_L is now defines the starting formula of A_{L^R}: $(\bar{z}_0 \wedge \bar{z}_1)$ (i.e. $\{q_0\} \implies \bar{x}_0 \wedge \bar{x}_1 \implies \bar{z}_0 \wedge \bar{z}_1$). Thus, for example, the resulting AFA deciding L^R can be defined as $A_{L^R} = (\{z_0, z_1\}, \{a\}, \delta', \bar{z}_0 \wedge \bar{z}_1, \emptyset)$ (δ' is given in table 3.3). □

Theorem 3.11. Some AFA with $\lceil \log(n+2) \rceil$ states has an equivalent DFA with at least 2^n states.

Proof. One of the languages we will give is the same as we used before: C_n, the language of binary strings that contain a 0 exactly n places from its end. The other one is ${C_n}^R$, the language of binary strings that contain a 0 exactly n places from its start. Clearly, a DFA with $n+2$ states can decide ${C_n}^R$ (i.e. a simple counter). We also know from the previous theorem that an AFA with $\lceil \log(n+2) \rceil$ states can recognize the reverse of ${C_n}^R$, which is actually C_n itself. We also know that C_n cannot be decided by a DFA with number of states less than 2^n by theorem 3.2. Thus, it can be concluded that an AFA with $\lceil \log(n+2) \rceil$ states has an equivalent DFA with at least 2^n states. □

We proved that the lower bound for the simulation of AFA by DFA is double exponential, just as the upper bound. This lower bound matches with the ones given in [7] and [18]. For unary regular languages, we will prove two more theorems to show that AFA-DFA conversion always has an exponential bound.

Theorem 3.12. Say a language L is decided by some AFA A_L with $\lceil \log n \rceil$ states. Then, the reverse language of L, namely L^R, is decided by a DFA D_{L^R} with n states.

Proof. The proof is simple actually. We only need to revert the process described in theorem 3.10, making a DFA deciding L^R by changing A_L accordingly. □

Theorem 3.13. A unary regular language L is decided by a minimal AFA with $\lceil \log n \rceil$ states if and only if L is decided by a minimal DFA with n states.

Proof. Firstly, the reverse language of a unary language is itself. For the first part (i.e. "only if" part), by theorem 3.12, if L is decided by a minimal AFA with $\lceil \log n \rceil$ states, then L is decided by a DFA with n states. Furthermore, by theorem 3.10, if L is decided by D_L with n states, then D_L should be minimal, since there does not exist an AFA deciding L with $n' < \lceil \log n \rceil$ states. One can use a similar way to show the second part (i.e. "if" part) also. Henceforth, the theorem is proved. □

3.5. Simulation of 2AFA by DFA

Simulation of 2-way AFA by DFA has been analyzed before, and an upper bound which is equal to 2^{n2^n} was introduced in 1978 by Ladner, Lipton, and Stockmeyer's paper "Alternating pushdown automata" [5]. We will now shortly prove the upper bound for the conversion of our 2AFA to pDFA.

Theorem 3.14. Every n-state 2AFA has an equivalent $2^{(n+1)2^n}$-state pDFA.

Proof. We suppose that a 2AFA $D = (Q, \Sigma_{\{\$,\#\}}, \delta, f_0, F)$ is given and $|Q| = n$.

Similar to the tables of 2DFA and 2NFA on words that are described in [16], we will define the notion of "table of 2AFA on a word w". However, unlike Kapoutsis, we will not directly construct a pDFA or DFA that simulates our D but we will show that there exists a finite number of different tables that can be used to separate words, just as states of a DFA do. First, we will give some important definitions:

Definition 3.5. Given a word w and a computation tree T on w, we recursively define the following: *T hits right into some CDNF f*,

- if every computation subtree of T, say T_i, hits right into some CDNF f_i such that $f = f_1 \wedge f_2 \wedge ... \wedge f_n$, for $1 \leq i \leq n$.
- if T consists of only one terminal node (h, x) such that $h = f$ and $x = |w| + 1$.

From the definition above, one can see that for T to hit right into some CDNF f, it must have a finite number K of terminal nodes denoted by $(g_i, |w| + 1)$ such that $f = g_1 \wedge g_2 \wedge g_3 \wedge ... \wedge g_K$.

Additionally, if a computation tree hits left, or hangs or loops inside the input (i.e. does not hit right), then we say it hits right into the CDNF 0 (i.e. "false"). Consequently, if for every possible computation tree $COMP_{D,h,p}(w)_i$ on w that hits some $G_i \neq 0$ (i.e. G_i is not a logical contradiction), we can say that "the full computation $FCOMP_{D,h,p}(w)$ on w hits right into G" where $G = G_1 \vee G_2 \vee ... \vee G_K$, $1 \leq i \leq K$. One can see that the number of different G_i should be finite.

Definition 3.6. Suppose the full computation of w on 2AFA D when started at the CDNF h on the first symbol of w, shortly denoted by $FCOMP_{D,h,1}(w)$, hits right into some CDNF G. Define the *table of D on w* to be the function

$$TABLE_D(w) = \tau : Q \cup \bot \rightarrow CDNF_Q \tag{3.12}$$

that satisfies $\tau(\bot) = G$ and for all $q \in Q$

- $\tau(q) = H$ if $FCOMP_{D,g,|w|}(w)$ hits right into some $H \neq 0$,
- $\tau(q) = 0$ if $FCOMP_{D,g,|w|}(w)$ does not hit right.

where g is a CDNF such that $g = q$.

One can see that there can be at most $2^{(n+1)2^n}$ different tables that are associated with words. Since the number of all words are infinite, by the pigeonhole principle, some words must share the same tables. Thus, we can group all words into $2^{(n+1)2^n}$ different groups at maximum. Now, we will show that given a table T_w for a word w, one can calculate the table $T_{w'}$ for the word $w' = wa$ where a is a symbol:

- For any $q \in Q$, since we know the transition function of our 2AFA D, we can calculate $FCOMP_{D,g,|wa|}(wa)$ by finding every possible computation tree that hits right into some CDNF G_i and producing the disjunction of them $G = G_1 \vee G_2 \vee ... \vee G_K$ where K is finite. Here, we do not show an efficient way to do that, but we simply say that G can be computed. Hence, $T_{w'}$ should have an entry $\tau_{w'}(q) = G'$ where G' is a CDNF sch that $G' = G$.
- For any $\perp$, since we know the transition function of our 2AFA D, we can calculate $FCOMP_{D,f_0,1}(wa)$ by finding every possible computation tree that hits right into some CDNF and producing the disjunction of them (similar to the previous case). Here, we do not show an efficient way to do that, but we simply say that such a calculation can be done. Hence, $T_{w'}$ should have an entry $\tau_{w'}(\perp) = H$ where H is the resulting CDNF equivalent to the disjunction of all found CDNFs that computation trees hit right into.

From these arguments, we can infer that a DFA with no more than $2^{(n+1)2^n}$ states can recognize the language D decides. Its starting state is actually a table in which every entry shows the starting CDNF of D and final state set consists of tables in which the first entry is always $\tau(\perp) = \eth$ such that every $FCOMP_{D,h,1}(\$w\#)$ hits right into some CDNF $\eth$ and $\eth$ evaluates to 1 (i.e. true) when 1 is substituted for every $q \in F$ in $\eth$ and 0 (i.e. false) is substituted for every $q \notin F$ in $\eth$. The theorem is proved. $\square$

Currently, we do not know a lower bound for simulation of 2AFA by DFA higher than 2^{2^n} which is actually the lower bound for AFA-to-DFA conversion theorem 3.11.

3.6. Simulation of PFA by DFA

In general, PFA are more powerful than DFA in language recognition, as stated by Rabin in [3]. However, it is proven that bounded-error PFA cannot recognize a language other than regular. We will show that there exists a DFA equivalent to every bounded-error PFA, analyzing and using the proof of Rabin [3] (which is actually designated for PFA with isolated cut-point, another way of describing PFA with error bounds).

Theorem 3.15. Let P be a bounded-error PFA with error margin ϵ. If P decides some language L, then there exists a DFA D that decides the same L. Moreover, if P has n states $\{q_1, q_2, ..., q_n\}$ with only q_n being an accepting state, then D can be chosen to have e states where

$$e \leq (1 + \frac{1}{\gamma})^{n-1} \tag{3.13}$$

where $\gamma = 0.5 - \epsilon$ and γ is called the radius of isolation of P [3].

Proof. Let $S = \{u_1, u_2, ..., u_e\}$ be the set of words that are pairwise inequivalent by $\equiv_L$. To find the DFA equivalent of P, we will show that e is finite and can be bounded.

We know that every word s has an associated computation ρ_s, the final probability distribution vector of s. Given two words $u_i, u_j \in S$ where $i \neq j$, if $u_i \not\equiv_L u_j$, then for some word w, $u_i w \in L$ and $u_j w \notin L$. This means that $p(u_i w) \geq 1 - \epsilon$ and $p(u_j w) \leq \epsilon$, giving $p(u_i w) - p(u_j w) \geq 1 - 2\epsilon = 2\gamma$. Thus, $\|\rho_{u_i w} - \rho_{u_j w}\| \geq 2\gamma$, since $\rho_{u_i w}[n] - \rho_{u_j w}[n] = p(u_i w) - p(u_j w) \geq 2\gamma$, where $\|x\|$ denotes the maximum of the absolute values of the entries of a given vector x and $\rho_{u_i w}[n]$ (resp. $\rho_{u_j w}[n]$) denotes the n^{th} entry of $\rho_{u_i w}$ (resp. $\rho_{u_j w}$). Furthermore, we can write $\rho_{u_i w}$ and $\rho_{u_j w}$ as $\rho_{u_i w} = W\rho_{u_i}$ and $\rho_{u_j w} = W\rho_{u_j}$ where

- W is an $n \times n$-dimensional column stochastic matrix that describes the computation of P on the suffix w of $u_i w$ and $u_j w$, and

- ρ_{u_i} and ρ_{u_j} denote the final probability distribution vectors associated with u_i and u_j, respectively.

This gives us the following two inequalities:

$$2\gamma \leq \|\rho_{u_i w} - \rho_{u_j w}\| = \|W\rho_{u_i} - W\rho_{u_j}\| = \|W(\rho_{u_i} - \rho_{u_j})\| = |W_g(\rho_{u_i} - \rho_{u_j})| \quad (3.14)$$

$$|W_g(\rho_{u_i} - \rho_{u_j})| \leq W_g|(\rho_{u_i} - \rho_{u_j})| = \sum_{k=1}^{n} W_g[k]|\rho_{u_i}[k] - \rho_{u_j}[k]| \leq \sum_{k=1}^{n} |\rho_{u_i}[k] - \rho_{u_j}[k]| \quad (3.15)$$

where W_g denotes the g^{th} row of W such that $|W_g(\rho_{u_i} - \rho_{u_j})| = \|W(\rho_{u_i} - \rho_{u_j})\|$, and $W_g[k]$, $\rho_{u_i}[k]$ and $\rho_{u_j}[k]$ denote the k^{th} entry of W_g, ρ_{u_i} and ρ_{u_j}, respectively.

Here, we have found something: If $u_i \not\equiv_L u_j$, then $\sum_{k=1}^{n} |\rho_{u_i}[k] - \rho_{u_j}[k]| = |\rho_{u_i} - \rho_{u_j}| \geq 2\gamma$. From now on, we will consider the probability distribution vectors ρ_{u_l} of $u_l \in S$ as points in Euclidean n-space. Thus, we can say that if two words u_i and u_j are pairwise distinguishable, then the rectilinear distance between their final probability distribution vectors (i.e. $\sum_{k=1}^{n} |\rho_{u_i}[k] - \rho_{u_j}[k]|$) is not smaller than 2γ.

Now, we will define a confiner set η_l for every ρ_{u_l}:

$$\eta_l = \{(x_1, x_2, ..., x_n) \mid \rho_{u_l}[i] \leq x_i,\ 1 \leq i \leq n,\ \sum(x_i - \rho_{u_l}[i]) = \gamma\}$$

Each η_l is a translate by ρ_{u_l}, of the set:

$$\eta = \{(x_1, x_2, ..., x_n) \mid 0 \leq x_i,\ 1 \leq i \leq n,\ \sum x_i = \gamma\}$$

η has an $(n-1)$-dimensional volume $V_{n-1}(\eta)$, and it is equal to $c\gamma^{n-1}$ for some constant c. Furthermore, for every $(x_1, x_2, ..., x_n) \in \eta_l$, since $\sum \rho_{u_l}[i] = 1$, it can be seen that

$\sum x_i = 1 + \gamma$, $0 \leq x_i$, $1 \leq i \leq n$. Thus, $\eta_l \subseteq \tau$, where

$$\tau = \{(x_1, x_2, ..., x_n) \mid 0 \leq x_i,\ 1 \leq i \leq n,\ \sum x_i = 1 + \gamma\}$$

One can see that given any two different sets η_i and η_j, and a point $(x_1, x_2, ..., x_n)$, the following conditions should not hold together

(i) $(x_1, x_2, ..., x_n) \in \eta_i$ and $(x_1, x_2, ..., x_n) \in \eta_j$,
(ii) $\rho_{u_i}[k] < x_k$ and $\rho_{u_j}[k] < x_k$,

for all $1 \leq k \leq n$. Otherwise, for all $1 \leq k \leq n$, we would have the following result:

$$|\rho_{u_i}[k] - \rho_{u_j}[k]| = |(\rho_{u_i}[k] - x_k) + (x_k - \rho_{u_j}[k])| < |x_k - \rho_{u_i}[k]| + |x_k - \rho_{u_j}[k]| \quad (3.16)$$

which gives the following contradiction:

$$2\gamma \leq \sum_{k=1}^{n} |\rho_{u_i}[k] - \rho_{u_j}[k]| < \sum_{k=1}^{n} |x_k - \rho_{u_i}[k]| + \sum_{k=1}^{n} |x_k - \rho_{u_j}[k]| = \gamma + \gamma = 2\gamma \quad (3.17)$$

Therefore, any two such different sets η_i and η_j are disjoint and considering their volumes, we have the following result:

$$ec\gamma^{n-1} = \sum_{i=1}^{e} V_{n-1}(\eta_i) \leq V_{n-1}(\tau) = c(1 + \gamma)^{n-1} \quad (3.18)$$

which means that $e \leq (1 + \frac{1}{\gamma})^{n-1}$. Thus, e is finite and there exists a DFA that decides L with no more than e states. □

The proof above can be generalized to the case where a PFA has h final states and in that case $e \leq (1 + \frac{h}{\gamma})^{n-1}$, as Rabin suggests. Now, we change our direction and prove that PFA can be more succinct than DFA when deciding regular languages:

Theorem 3.16. Let E_n be a URL such that $E_n = \{a^k \mid k \geq n\}$. Any DFA deciding E_n has at least $n + 1$ states whereas there exists a two-state PFA that decides E_n.

Proof. Our language E_n actually describes the set of integers that are equal to or bigger than some given n. Clearly, an $n+1$-state UDFA with one accepting state (which is the $n+1^{th}$ state) decides E_n. A UDFA cannot have fewer than $n+1$ states, otherwise it can accept strings that are shorter than n. Now, we give the PFA that can decide E_n.

Suppose there exist two states in our PFA: The starting state q_0 and one final state q_1 (q_0 is not final). We can define all four possible transitions between states as follows:

(i) $q_0 \to q_0$ with some probability α,
(ii) $q_0 \to q_1$ with some probability $1-\alpha$,
(iii) $q_1 \to q_0$ with probability 0,
(iv) $q_1 \to q_1$ with probability 1,

The computation principle of our PFA is absurdly simple: In every step it goes to the accepting state with $1-\alpha$ and stay in the starting state with α. Since we use q_1 as a "sink" accepting state; the longer the string, the higher its acceptance probability. Precisely, the acceptance probability of a word a^k is $p(a^k) = 1-\alpha^k$. Thus, by considering the conditions that $p(a^{n-1}) \leq \epsilon$ and $p(a^n) \geq 1-\epsilon$, we can provide a PFA that decides E_n with error margin ϵ, which satisfies the following two inequalities

$$\epsilon \geq p(a^{n-1}) = 1-\alpha^{n-1} \quad \text{and} \quad \epsilon \geq 1-p(a^n) = \alpha^n \tag{3.19}$$

Notice that ϵ is not constant and changes with α and n. Thus, even though we save much states by using PFA instead of DFA to decide E_n, we cannot fix an error margin that is same for every n, using only two states. Later in theorem 4.4, we will give a family of languages such that the deciding bounded-error PFA perform statewise exponentially better than their DFA counterparts. □

3.7. Simulation of QFA by DFA

In general, QFA are more powerful than DFA in language recognition [14], [20]. However, it is proven that bounded-error QFA cannot recognize a language other than regular. In this section, we will prove three theorems. First, we will address an upper bound on simulation of a special kind of bounded-error QFA by DFA; QFA that only has orthogonal matrices (i.e. square, not rectangular, matrices) as superoperators. Then, we will do that for general bounded-error QFA; QFA that has rectangle matrices as superoperators. Finally, we will address a lower bound for the first case.

First of all, we explain more about how quantum systems behave and evolve.

In this part, we will call QFA having only orthogonal matrices as superoperators as "orthogonal QFA" or shortly OQFA. To understand more about OQFA, we will give another representation for the computation and acceptance of OQFA.

Let P be a OQFA where $P = (\{q_1, q_2, ..., q_n\}, \Sigma, \{\varepsilon_\sigma\}_{\sigma\in\Sigma}, q_1, F)$.

Definition 3.7. Suppose some word $w = w_1w_2...w_r$ where $|w| = r$ is given and every w_i is a symbol where $1 \leq i \leq r$. The *computation ϕ_w of OQFA P on word w* is the vector ϕ_w of size $n \times 1$ resulted after reading $w = w_1w_2...w_r$ and recursively defined as

$$\phi_{uw_j} = E_{w_j}\phi_u \tag{3.20}$$

where $u = w_1w_2...w_{j-1}$. ϕ_ϵ is the *starting quantum distribution* and a $n\times 1$-dimensional single-entry vector in which the first entry is 1. ϕ_w is called as the *final quantum distribution of w*.

Definition 3.8. OQFA P *accepts* a word w with probability $p'(w) = \sum_{q_i\in F} |\phi_w[i]|^2$ where $\phi_w[i]$ denotes i^{th} entry of ϕ_w.

As one can see these computation and acceptance definitions of OQFA are slightly different than those of (general) QFA. However, both OQFA and QFA definitions are

actually equivalent in essence. We can give a quick conversion protocol to change OQFA definitions to QFA definitions, using our OQFA P and word w as example:

- $\phi_w \phi_w^\dagger = \rho_w$ (ρ_w is the final quantum distribution matrix of w, definition 2.38),
- $p'(w) = p(w) = \sum_{q_i \in F} \phi_w[i][i]$ ($p(w)$ is the acceptance probability of w, definition 2.39).

In this representation, we use vectors instead of matrices to show quantum distribution over states, similar to the PFA case. However, because of quantum nature of states, it is not exactly like in the PFA case. All entries in a probability distribution vector are chosen to be real numbers in the interval $[-1, 1]$ and Euclidean norm of a vector should be 1 (i.e. squares of entries add upto 1). Thus, when we give a vector $V = (p_1 \;\; p_2 \;\; ... \;\; p_n)^T$, it practically means that we can observe a state q_i with p_i^2 probability, as quantum nature of states dictates us.

Our OQFA P reads as follows: Given a word w, P starts its computation as the given vector $\rho_{w,0}$ which is a way of saying that it starts on the starting state. In the j^{th} step, as P reads some symbol a, an operation which can be represented as a matrix operation on the previous computed matrix $\phi_{w,j-1}$ with E_a. Such an operation changes the current quantum distribution over all states. At the end, it accepts w with the probability $p(w) = \sum_{q_i \in F} |\phi_w[i]|^2$ which is a way of saying that the computation ends on the final states with a total probability of $p(w)$.

Now, we will use this representation of OQFA in the following theorem.

Theorem 3.17. Let P be a bounded-error n-state OQFA with error margin ϵ. Let P has a state set $Q = \{q_1, ..., q_n\}$, an accepting state set F, and f accepting states. If P decides some language L, then there exists an e-state DFA D that decides the same L where

$$e \leq (\sqrt{1 + \frac{4f^2}{\gamma^2}})^{n-1} \tag{3.21}$$

where $\gamma = 0.5 - \epsilon$ and γ is called the radius of isolation of P.

Proof. We will prove this theorem in a similar way we prove the theorem 3.15. Let $S = \{u_1, u_2, ..., u_e\}$ be the set of words that are pairwise inequivalent by $\equiv_L$. To find the DFA equivalent of P, we will show that e is finite and can be bounded.

Given two words $u_i, u_j \in S$ where $i \neq j$, if $u_i \not\equiv_L u_j$, then for some word w, $u_i w \in L$ and $u_j w \notin L$. Thus, we know that the acceptance probabilities of $u_i w$ and $u_j w$ are $p(u_i w) \geq 1 - \epsilon$ and $p(u_j w) \leq \epsilon$ respectively, giving $p(u_i w) - p(u_j w) \geq 1 - 2\epsilon = 2\gamma$. Also, we know that every word s has an associated computation ϕ_s, the final quantum distribution vector of s. Thus, for u_i, u_j, $u_i w$, and $u_j w$, we have ϕ_{u_i}, ϕ_{u_j}, $\phi_{u_i w}$ and $\phi_{u_j w}$ where $\phi_{u_i w} = M_w \phi_{u_i}$ and $\phi_{u_j w} = M_w \phi_{u_j}$ for some orthogonal matrix M_w that describes the computation of P on the suffix w of $u_i w$ and $u_j w$. Thus, one can see that

$$2\gamma \leq p(u_i w) - p(u_j w) = \sum_{q_k \in F} (\phi_{u_i w}[k])^2 - (\phi_{u_j w}[k])^2 = \sum_{q_k \in F} ((M_w \phi_{u_i})[k])^2 - ((M_w \phi_{u_j})[k])^2 \tag{3.22}$$

Say we define $\|\phi_{u_i} - \phi_{u_j}\|$ as $\|\phi_{u_i} - \phi_{u_j}\| = \sqrt{\sum_{k=1}^{n} |\phi_{u_i}[k] - \phi_{u_j}[k]|^2}$. Now, we will show that $\frac{\gamma}{f} \leq \|\phi_{u_i} - \phi_{u_j}\|$:

$$\begin{aligned}
2\gamma &\leq \sum_{q_k \in F} ((M_w \phi_{u_i})[k])^2 - ((M_w \phi_{u_j})[k])^2 \\
&= \sum_{q_k \in F} (m_k \phi_{u_i})^2 - (m_k \phi_{u_j})^2 \\
&= \sum_{q_k \in F} (m_k \phi_{u_i} + m_k \phi_{u_j})(m_k \phi_{u_i} - m_k \phi_{u_j}) \\
&= \sum_{q_k \in F} (m_k(\phi_{u_i} + \phi_{u_j}))(m_k(\phi_{u_i} - \phi_{u_j})) \\
&\leq \sum_{q_k \in F} (\|m_k\| \|\phi_{u_i} + \phi_{u_j}\|)(\|m_k\| \|\phi_{u_i} - \phi_{u_j}\|) \\
&= f \|\phi_{u_i} + \phi_{u_j}\| \|\phi_{u_i} - \phi_{u_j}\|
\end{aligned} \tag{3.23}$$

where m_k denotes the k^{th} row of M_w. Using the facts that $2\gamma \leq f\|\phi_{u_i}+\phi_{u_j}\|\|\phi_{u_i}-\phi_{u_j}\|$ and $|\phi_{u_i}+\phi_{u_j}\| \leq 2$, we can infer that $\frac{\gamma}{f} \leq \|\phi_{u_i}-\phi_{u_j}\|$. Furthermore, similar to the case in the theorem 3.15, we can represent ϕ_{u_i} and ϕ_{u_j} as points in Euclidean n-space. Thus, we can say that if two words u_i and u_j are pairwise distinguishable, then the Euclidean distance between these points (i.e. $\sqrt{\sum_{k=1}^{n}|\phi_{u_i}[k]-\phi_{u_j}[k]|^2}$) is not smaller than $\frac{\gamma}{f}$. Just as in the theorem 3.15, we will define a confiner set η_l for every ρ_{u_l}:

$$\eta_l = \{(x_1,...,x_n) \mid 0 \leq \rho_{u_l}[i] \leq x_i \text{ or } 0 \geq \rho_{u_l}[i] \geq x_i,\ 1 \leq i \leq n,\ \sum x_i^2 - (\rho_{u_l}[i])^2 = \frac{\gamma^2}{4f^2}\}$$

Each η_l is a translate by ρ_{u_l}, of the set:

$$\eta = \{(x_1,x_2,...,x_n) \mid 1 \leq i \leq n,\ \sum x_i^2 = \frac{\gamma^2}{4f^2}\}$$

η has an $(n-1)$-dimensional volume $V_{n-1}(\eta)$, and it is equal to $c(\sqrt{\frac{\gamma^2}{4f^2}})^{n-1}$ for some constant c. Furthermore, for every $(x_1,x_2,...,x_n) \in \eta_l$, since $\sum(\rho_{u_l}[i])^2 = 1$, it can be seen that for $1 \leq i \leq n$, $\sum x_i^2 = \frac{\gamma^2}{4f^2}+1$. Thus, $\eta_l \subseteq \tau$, where

$$\tau = \{(x_1,x_2,...,x_n) \mid 1 \leq i \leq n,\ \sum x_i^2 = 1+\frac{\gamma^2}{4f^2}\}$$

One can see that given any two different sets η_i and η_j, and a point $(x_1,x_2,...,x_n)$, the following conditions should not hold together

(i) $(x_1,x_2,...,x_n) \in \eta_i$ and $(x_1,x_2,...,x_n) \in \eta_j$,
(ii) $0 \leq |\rho_{u_i}[k]| < |x_k|$ and $0 \leq |\rho_{u_j}[k]| < |x_k|$,

for all $1 \leq k \leq n$. Otherwise, for all $1 \leq k \leq n$, we would have the following result:

$$
\begin{aligned}
&|\rho_{u_i}[k] - \rho_{u_j}[k]| < |x_k - \rho_{u_i}[k]| + |x_k - \rho_{u_j}[k]| \\
&|\rho_{u_i}[k] - \rho_{u_j}[k]|^2 < |x_k - \rho_{u_i}[k]|^2 + 2|x_k - \rho_{u_i}[k]||x_k - \rho_{u_j}[k]| + |x_k - \rho_{u_j}[k]|^2 \\
&|\rho_{u_i}[k] - \rho_{u_j}[k]|^2 < 2(|x_k - \rho_{u_i}[k]|^2 + |x_k - \rho_{u_j}[k]|^2) \\
&|\rho_{u_i}[k] - \rho_{u_j}[k]|^2 < 2(|x_k^2 - (\rho_{u_i}[k])^2| + |x_k^2 - (\rho_{u_j}[k])^2|)
\end{aligned}
\tag{3.24}
$$

This gives us the following:

$$
\frac{\gamma^2}{f^2} \leq \sum_{k=1}^{n} |\rho_{u_i}[k] - \rho_{u_j}[k]|^2 < 2(\sum_{k=1}^{n} |x_k^2 - (\rho_{u_i}[k])^2| + \sum_{k=1}^{n} |x_k^2 - (\rho_{u_j}[k])^2|) = 2(\frac{\gamma^2}{4f^2} + \frac{\gamma^2}{4f^2}) \tag{3.25}
$$

Thus, there is a contradiction that $\frac{\gamma^2}{f^2} < \frac{\gamma^2}{f^2}$. Therefore, any two such different sets η_i and η_j are disjoint and considering volumes, we have the following result:

$$
ec(\sqrt{\frac{\gamma^2}{4f^2}})^{n-1} = \sum_{i=1}^{e} V_{n-1}(\eta_i) \leq V_{n-1}(\tau) = c(\sqrt{1 + \frac{\gamma^2}{4f^2}})^{n-1} \tag{3.26}
$$

which means that $e \leq (\sqrt{1 + \frac{4f^2}{\gamma^2}})^{n-1}$. Thus, e is finite and there exists a DFA that decides L with no more than e states. □

One can see that for a fixed error bound, the result above actually says that upper bound for simulation of OQFA by DFA is $2^{O(n)}$, compatible with the result given by Ambainis and Yakaryilmaz in [14]. Now, we will move onto general QFA.

Before going further, we will state another important thing about quantum computation. If our knowledge of the current superposition of a quantum system is certain, then the system is said to be in "pure state". Sometimes, we only know that the system is in some pure state v_i with probability p_i for $i > 1$, and with the probabilities adding up to 1, in this case, the system is said to be in a "mixed state". One can see that the

computation procedure of OQFA does not allow a quantum system to be in a mixed state. The computation procedure of general QFA, on the other hand, allows that. Thus, to find the upper bound for simulation of QFA by DFA, we need to consider this situation. Now, we prove the following theorem about QFA and regular languages.

Theorem 3.18. The languages recognized by QFA with bounded error are exactly the regular languages.

Proof. It is known that QFA can recognize regular languages by simulating DFA. We will prove this part in lemma 4.20, by simulating PFA with QFA.

For the other part, now we will prove that bounded-error QFA can only recognize regular languages, in a similar way we prove the theorem 3.15. Suppose a QFA P with n states and ϵ error bound decides a language L. Let $S = \{u_1, u_2, ..., u_e\}$ be the set of words that are pairwise inequivalent by $\equiv_L$. We will show that e is finite.

Given two words $u_i, u_j \in S$ where $i \neq j$, if $u_i \not\equiv_L u_j$, then for some word w, $u_i w \in L$ and $u_j w \notin L$. Thus, we know that the acceptance probabilities of $u_i w$ and $u_j w$ are $p(u_i w) \geq 1 - \epsilon$ and $p(u_j w) \leq \epsilon$ respectively, giving $p(u_i w) - p(u_j w) \geq 1 - 2\epsilon$. We also know that

$$p(u_i w) - p(u_j w) = \sum_{q_k \in F} \rho_{u_i w}[k][k] - \sum_{q_k \in F} \rho_{u_j w}[k][k] \tag{3.27}$$

where F, $\rho_{u_i w}[k][k]$, and $\rho_{u_j w}[k][k]$ respectively denote the set of final states of P and the entry in the k^{th} row and k^{th} column of $\rho_{u_i w}$ and $\rho_{u_j w}$ ($\rho_{u_i w}$ and $\rho_{u_j w}$ are the computations associated with $u_i w$ and $u_j w$). We also have the following:

$$\|\rho_{u_i w} - \rho_{u_j w}\|_{tr} \geq \sum_{q_k \in F} \rho_{u_i w}[k][k] - \sum_{q_k \in F} \rho_{u_j w}[k][k] \geq 1 - 2\epsilon \tag{3.28}$$

where $\|A - B\|_{tr} = Tr(\sqrt{(A-B)(A-B)^T})$ denotes the TR-distance between density matrices A and B ($Tr(M)$ denotes the trace of a square matrix M). Furthermore,

$\|\rho_{u_i} - \rho_{u_j}\|_{tr} \geq \|\rho_{u_i w} - \rho_{u_j w}\|_{tr}$ since evolving two different density matrices ρ_{u_i} and ρ_{u_j} (which are actually results of applying different superoperators on the initial density matrix) by using the same superoperators on them (i.e. the computation process from ρ_{u_i} and ρ_{u_j} to $\rho_{u_i w}$ and $\rho_{u_j w}$) does not increase the TR-distance between them. Thus, for every pairwise inequivalent u_i and u_j, the TR-distance between their associated computations ρ_{u_i} and ρ_{u_j} is not smaller than $1 - 2\epsilon$. If e is infinite, then there must exist two words $u_t \in S$ and $u'_t \in S$ such that the TR-distance between their associated computations ρ_{u_t} and $\rho_{u'_t}$ is not bigger than $1 - 2\epsilon$, contradicting our argument. Therefore, e should be finite and L must be regular. Hence, the languages recognized by QFA with bounded error are exactly the regular languages. □

Theorem 3.19. If a language L is decided by an n-state QFA with bounded-error, then it can also be decided by a $2^{O(n^2)}$-state DFA [14].

Proof. If L is decided by an n-state QFA with bounded-error ϵ, then we know that L is regular. Thus, there exists a set of words $S = \{u_1, u_2, ..., u_e\}$ such that they are all pairwise inequivalent by $\equiv_L$ and e is finite. We will bound e.

We know that a density matrix describes a quantum system in a mixed state. Thus, in any step, a QFA with n states can be in a mixed state ρ in n dimensions which can be represented as a mixture (p_k, v_k) of at most n pure states where p_k denotes the probability of being in some pure state v_k:

$$\rho = \sum_{k=1}^{m} p_k v_k v_k^T \tag{3.29}$$

where $m \leq n$ and v_k^T denotes the transpose of v_k. Thus, we can approximate each of v_k by a state v'_k from an ε-net for the unit sphere in n-dimensions (an ε-net is a set S such that for any v, there exists a $v' \in S$ such that $\|v - v_k\| \leq \varepsilon$). We will come to this point later.

Generating an ε-net is actually something similar to what we have done in the previous proofs for the theorems 3.17 and 3.15, and we know that one can create such

an ε-net with a size of $2^{O(n)}$ states (states being a set of vectors represented by points that are enclosed in a space with volume of $2^{O(n)}$). Thus, for a given set of pure states $\{v_1, v_2, ..., v_n\}$, we have $(2^{O(n)})^n = 2^{O(n^2)}$ choices that can be taken from our ε-net that approximate them. Furthermore, we can also approximate $\{p_1, p_2, ..., p_n\}$ by a state from another ε-net which has a size of $2^{O(n)}$ states. We will describe this approximation concept in the following argument:

Say that two words $u_i \in S$ and $u_j \in S$ are pairwise inequivalent. Thus, given their associated computations $\rho_{u_i} = \sum_{k=1}^{n_1} p_{1k} v_{1k} v_{1k}^T$ and $\rho_{u_j} = \sum_{k=1}^{n_2} p_{2k} v_{2k} v_{2k}^T$ where $n_1 \leq n$ and $n_2 \leq n$, the TR-distance between them is $\|\rho_{u_i w} - \rho_{u_j w}\|_{tr} \geq 1 - 2\epsilon$. By using our ε-nets for pure states and probabilities, we can approximate ρ_{u_i} and $\rho_{u_j} =$ by $\hat{\rho}_{u_i} = \sum_{k=1}^{n_2} \hat{p}_{1k} \hat{v}_{1k} \hat{v}_{1k}^T$ and $\hat{\rho}_{u_j} = \sum_{k=1}^{n_2} \hat{p}_{2k} \hat{v}_{2k} \hat{v}_{2k}^T$ such that $\|\hat{\rho}_{u_i w} - \hat{\rho}_{u_j w}\|_{tr} \geq 1 - 2\epsilon$. Hence, by using a $2^{O(n^2)}$-state DFA, we can decide L, completing our proof.

This proof is based on the proof that is given for Theorem 4.6 of [14]. □

Theorem 3.20. For any prime p, say B_p is the language such that $B_p = \{a^{kp} \mid k \in \mathbb{N}\}$. The minimal DFA that decides B_p has p states whereas there exists a bounded-error OQFA which decides B_p with $O(\log p)$ states.

Proof. It is proven in the following chapter, in theorem 4.12. □

This theorem suggests a lower bound for the simulation of bounded-error n-state OQFA by DFA, and it is $O(2^n)$ which matches with the upper bound given in the theorem 3.17. Currently, we do not know if there exists a lower bound for the simulation of n-state (general) QFA by DFA, that matches with the upper bound given in the theorem 3.19. This concludes the third chapter.

4. SUCCINCT FINITE AUTOMATA FOR THREE DIFFERENT FAMILIES OF REGULAR LANGUAGES

4.1. The Language Class A_m and Its Deciding Automata

In this section, we will introduce a unary regular language shortly denoted by A_m which consists of only 1 member with length m. Using $\Sigma = \{a\}$ as our alphabet, our language is $A_m = \{a^m\}$. Now, we will construct a DFA that recognizes A_m and prove that it is actually the smallest in terms of number of states it has.

The DFA D_{A_m} that recognizes A_m is $D_{A_m} = (Q, \Sigma, \delta, q_0, q_m)$ and its components are $Q = \{q_0, q_1, q_2, ..., q_m, q_{m+1}\}$, $\Sigma = \{a\}$, $\delta(q_{m+1}, a) = q_{m+1}$, and $\delta(q_i, a) = q_{i+1}$ for every i such that $0 \leq i \leq m$. Here, q_0 is the starting state and q_m is the one and only final state. As we can see, D_{A_m} has $m + 2$ states. Next, we will shortly show why it decides A_m.

When a string s is given, D_{A_m} computes on s in a way that for every symbol a it reads, it goes into the state q_{i+1} from q_i ($i = m + 1$ is an exception); so actually, we are using our states as a counter which can effectively count up to $m + 1$. If $|s| < m$, then the computation of s will end in a state that is not a final state. If $|s| > m$, then the computation will end in q_{m+1} which also is not a final state. Thus, only if $s = a^m$, it will be accepted. Hence, $m + 2$ states are sufficient. Now, we will prove that D_{A_m} is the smallest in terms of the number of states it requires.

Theorem 4.1. D_{A_m} is the smallest DFA which decides A_m.

Proof. Suppose that a DFA with $k < m + 2$ states can decide A_m. But this means that there is an accepting state q_{acc} that $q_{acc} = q_i$ where $0 \leq i \leq m$ (assuming states are enumerated in an increasing manner, just like in the case of D_{A_m}). q_{acc} could not be in the periodic part of this DFA since that would imply this DFA decides a periodic language. Thus, q_i is not in the periodic part and $i < m$ (q_m is always in the

periodic part in such a setting). That means some word $a^i \neq a^m$ is accepted, giving a contradiction. Hence, $m + 2$ states are necessary. □

As a side note, if we have a pDFA instead of our complete DFA model, $m + 1$ states will be necessary and sufficient. Unlike D_{A_m}, we do not need a periodic part consisting of the state q_{m+1}. pDFA model allows us to have only $m + 1$ states where $\delta(q_m, a)$ and $\delta(q_{m+1}, a)$ are not defined in the transition function. When discussing about NFA, 2DFA, and 2NFA, we will use this pDFA case to bring out similarities.

Now, we will show that the smallest 2NFA deciding A_m needs $m + 1$ states. By doing that, we will come into a conclusion that 2-way and/or nondeterminism does not give us state advantage when deciding A_m (i.e. the NFA, 2DFA, and 2NFA deciding A_m need at least $m + 1$ states, just like pDFA does).

Theorem 4.2. Every 2NFA deciding A_m needs at least $m + 1$ states.

Proof. Suppose a 2NFA decider N' for A_m has $m' < m + 1$ states. Take an accepting computation c. By definition of acceptance, this c should start in the left endmarker and visit the right endmarker. We can decompose c into segments where each segment starts and ends when an endmarker is reached. There are 2 types of segments: (i) "starting from left endmarker and ending in right endmarker" or "starting from right endmarker and ending in left endmarker" which are shortly called as "traversal"s and (ii) "starting from left endmarker and ending in left endmarker" or "starting from right endmarker and ending in right endmarker" which are shortly called as "u-turn"s. So, if N' has m' states, then in every traversal t, there exists a state q that is visited repeatedly (at least twice) first on the i^{th} position and then on the $i + j^{th}$ position of the input a^m for some $i > 0$ and $j > 0$. Then, for every $k > 0$, we can pump the traversal with length kj, meaning there exists a traversal t' acting similar to t on the input a^{m+kj} and does the following:

If q is visited first on the i^{th} position and then on the $i + j^{th}$ position in t, then there is a segment of computation c' in t decribing the part of computation that starts

from the i^{th} position and ends in the $i+j^{th}$ position. Suppose t consists of three parts: The part l_1 from the beginning of t to the start of c', c', and the part l_2 from the end of c' to the end of t. t' simulates l_1, then simulates c' k times, and finally simulates l_2. That way, we can keep the beginning state and ending state of t as same in t' and hence, we can pump t with kj.

Say there exist such js in all traversals $t_1, t_2, ..., t_n$ of c, namely $j_1, j_2, ..., j_n$, then, we can pump the whole computation with the length $J = j_1 j_2 ... j_n$. Hence, the word a^{m+J} can be accepted by N' by an accepting computation which imitates c: traversals are replaced by J-pumped traversals and u-turns remain the same. However, by definition, N' should not accept any string other than a^m, so such N' gives a contradiction. Thus, the 2NFA which decides A_m has at least $m+1$ states. □

The proof above is similar to the one that is given in Birget's paper [9]. We know that 2NFA are the most generalized one among DFA, NFA, 2DFA and 2NFA, hence, considering the result we obtained above, we can state the following:

Corollary 4.2.1. Considering state-wise efficiency, NFA, 2DFA, and 2NFA are not more concise than pDFA while deciding A_m.

Furthermore, as a side note, one can observe that the theorem 4.2 can be generalized to finite unary regular languages. To put it in other way, we can say that a 2NFA needs at least $m+1$ states to decide a finite unary regular language A'_m where A'_m is a set of strings in which the longest string's length is m. One can show its proof in a way very similar to that of theorem 4.2, the main argument is that one of the accepting computation of the longest string can be pumped in a way that a much longer string can be also accepted by the same 2NFA. Now, we turn our direction to AFA.

Theorem 4.3. The minimal AFA that decides A_m has $\lceil \log(m+2) \rceil$ states.

Proof. By theorem 3.13, we know that every n-state minimal DFA has an equivalent minimal $\lceil \log n \rceil$-state AFA. D_{A_m} has $m+2$ states, thus the minimal AFA that decides

A_m has $\lceil \log(m+2) \rceil$ states. Hence, the theorem is proved. □

We do not know if 2AFA can outperform AFA when deciding A_m. Now, we will turn to PFA and show that bounded-error PFA can perform statewise better than DFA:

Theorem 4.4. For any $\epsilon > 0$, there exists a bounded-error PFA that decides A_m with $O(\frac{\log^2 m}{\log \log m})$ states and ϵ error margin.

Proof. Our proof is based on the one given in Ambainis and Freivalds's 1998 paper [10].

Here, the PFA deciding B_m has $r = O(\frac{\log m}{\log \log m})$ groups of states. Every group i has P_i states that form a cycle where a cycle is disjoint from other cycles. Denote $P_1 < P_2 < ... < P_r$ as the first r prime numbers (we will give a more precise r later in the proof). One can calculate that $P_r = O(r \log r) = O(\log m)$. To put in other way, $O(\frac{\log m}{\log \log m})$ group of states are chosen and every group is actually a cycle of $O(\log m)$ states, making a total of $O(\frac{\log^2 m}{\log \log m})$ states. Also, in any group of states $G_i = \{q_1^i, q_2^i, ..., q_{P_i}^i\}$, the only accept state is the state q_j^i where $m \equiv j \pmod{P_i}$. Furthermore, there exists another sink rejecting state where every other state has a transition to that state with some small probability c. Moreover, the starting distribution is different than usual, meaning that instead of using 1 state as the starting state with probability 1 at the beginning, we use the starting states of every group as one of r starting states, each having the starting probability $\frac{1}{r}$ (i.e. instead of using $(1 \;\; 0 \;\; 0 \;\; ... \;\; 0)^T$ vector, we use

$$(\frac{1}{r} \;\; 0 \;\; ... \;\; 0 \;\; \frac{1}{r} \;\; 0 \;\; ... \;\; 0 \;\; ... \;\; ... \;\; \frac{1}{r} \;\; 0 \;\; ... \;\; 0 \;\; 0)^T$$

vector at the beginning and starting probabilities $\frac{1}{r}$ are only for the r start states). It can be thought as a preprocessed probabilistic branching to the aforementioned r states.

The computation procedure of this PFA is as follows: Given some word a^x, in the i^{th} probabilistic branch, it checks whether $x \pmod{P_i} \equiv m \pmod{P_i}$ for P_i. In [10], it

is said that Freivalds had proved in the article [24] that with the use of such r groups of P_i-cyclic states and the sink rejecting state, the informally constructed PFA given above decides A_m with a low margin of error; behaving like a "probabilistic clock" that counts upto m. In his words: "The number of used primes suffices to assert that, for every input of length less than m, most of primes give remainders different from the remainder of m modulo P_i. The small probability (c) is chosen to have the rejection probability high enough for every input length M such that both $M \neq m$ and an ϵ-fraction of all the primes used have the same remainders modulo P_i as m." Practically, for some integer M', he states that the acceptance probability of a^m is significantly higher than acceptance probability of any other a^M since:

- For every $M > M'$, $p(a^M) < p(a^m)$: The sink rejecting state gathers "much" rejecting probability from other states as M grows large (and secondly, only a small number of our groups accept a^M).
- For every $M < M'$, $p(a^M) < d < \frac{1}{2} < p(a^m)$ for some constant d: Only a small number of our groups accept a^M (and secondly, the sink rejecting state has some rejecting probability that grows as M grows).

Henceforth, there exists a bounded-error PFA that decides A_m with $O(\frac{\log^2 m}{\log \log m})$ states. For further information about the numbers M' and d, here is an analysis, given in Balodis' paper [25]:

Say $P_1 < P_2 < ... < P_r$ are the first r primes and the product of first l of them $P_1P_2...P_l \geq m$. Then, for all $n < P_1P_2...P_l$, at most $l-1$ of the following r modular equivalences are satisfied given that $M \neq m$:

$$M \equiv m \pmod{P_i} \quad \text{for all } i \leq r \quad \text{(by the Chinese Remainder Theorem)} \tag{4.1}$$

We choose l to be the minimal number such that $P_1P_2...P_l \geq 2m$ and r to be $3l$. By using these r primes as our primes, we can achieve the following results about acceptance probabilities:

- $M < 2m$ and $M \neq m$: $p(a^M) \leq \frac{l-1}{r} < \frac{l}{3l} = \frac{1}{3}$
- $M = m$: $p(a^M) = 1 - E(a^m) = 1 - (1 - (1-c)^m) = (1-c)^m$
- $M \geq 2m$: $p(a^M) \leq 1 - E(a^{2m})$

where $E(a^M) = 1 - (1-c)^M$ is the probability that a^M is in the sink rejecting state after M steps. Suppose we choose $c = 1 - \sqrt[m]{\frac{3}{5}}$. We get

- $p(a^M) < \frac{1}{3}$ for $M < 2m$ and $M \neq m$,
- $p(a^m) = \frac{3}{5}$,
- $p(a^M) < 1 - E(a^{2m}) = \frac{9}{25} < \frac{2}{5}$ for $M \geq 2m$.

Henceforth, we construct a bounded-error PFA that decides A_m with $O(\frac{\log^2 m}{\log \log m})$ states for some error margin, which is in this case equal to $\frac{2}{5}$.

We want to highlight that choosing l to be the minimal number such that $P_1 P_2 ... P_l \geq 2m$ and r to be $3l$ is very important. We know that

$$2m \leq P_1 P_2 ... P_l < P_1 P_2 ... P_r = e^{(1+o(1))r \log r} \tag{4.2}$$

and this guarantees that $r \log r = O(\log m)$ and so $r = O(\frac{\log m}{\log \log m})$. Notice that the product $P_1 P_2 ... P_r$ is known as the r^{th} primorial $P_r\#$, and asymptotically, primorials $P_n\#$ grow according to $P_n\# = e^{(1+o(1))n \log n}$. □

Currently, we do not know that for any error margin $\epsilon > 0$, there exists a bounded-error PFA that decides A_m with $O(\log m)$ states. Now, we turn to QFA and show that QFA can perform (statewise) better than DFA, even better than PFA:

Theorem 4.5. There exists a bounded-error QFA that decides A_m with $O(\log m)$.

Proof. Because of the similarity between the counting natures of A_m and B_m, we will utilize the result of theorem 4.12. Thus, for the modular counting operation for any prime P_i, we only need $O(\log P_i)$ states, (instead of P_i states as was in the PFA case)

while reconstructing the PFA given above in QFA format. Since $P_i = O(\log m)$, in this case, the number of states used to recognize A_m should be equal to

$$O(\frac{\log m}{\log\log m}\log P_i) = O(\frac{\log m}{\log\log m}\log\log m) = O(\log m) \tag{4.3}$$

Thus, there exists a bounded-error QFA that decides A_m with $O(\log m)$ states. □

4.2. The Language Class B_m and Its Deciding Automata

In this section, we will introduce a unary regular language shortly noted by B_m which consists of members with integer multiples of length m. Actually, $B_m = \{a^{km} \mid k \in \mathbb{N}\}$. Now, we will construct a DFA that recognizes B_m.

The DFA D_{B_m} recognizing B_m is $D_{B_m} = (Q, \Sigma, \delta, q_0, q_{m-1})$ and its components are $Q = \{q_0, q_1, q_2, ..., q_{m-1}\}$, $\Sigma = \{a\}$, $\delta(q_{m-1}, a) = q_0$, and $\delta(q_i, a) = q_{i+1}$ where $0 \leq i \leq m-2$. Here, q_0 is the starting and the only final state. As we can see, D_{B_m} has m states. Next, we will shortly show why it decides B_m.

When a string s is given, D_{B_m} computes on s in a way that for every symbol a it reads, it goes into the state q_{i+1} from q_i (q_{m-1} is an exception); so actually, we are using our states as a counter which can effectively count from 0 to $m-1$ in modulo m. If $|s| \neq km$, then the computation of s will end in a rejecting state. If $|s| = km$ for any $k \in \mathbb{N}$, then the computation will end in q_{m-1}, the accepting state. Hence, m states are sufficient. Now, we will prove that D_{B_m} is the smallest in terms of the number of states it requires. In fact, we will prove that D_{B_m} is the smallest NFA which decides B_m:

Theorem 4.6. D_{B_m} is the smallest NFA which decides B_m.

Proof. Suppose that an NFA N' with $m' < m$ states can decide B_m. Since N' decides B_m, it accepts a^m. By the pigeonhole principle, this means that during the acceptance of a^m, N' visits some state q at least twice. Say the first two of those visitings are on

the x^{th} and $(x+y)^{th}$ symbols of a^m where $x > 0$ and $y > 0$. Such a situation means that a^{m-y} can also be accepted by N', giving a contradicition. Hence, m states are necessary. □

In order to show the smallest 2DFA and 2NFA deciding B_m, we consider the prime factorization of m. Writing with its prime factors, say $m = p_1^{k_1} p_2^{k_2} ... p_n^{k_n}$. We will show that a 2DFA deciding B_m with $p = p_1^{k_1} + ... + p_n^{k_n}$ states.

Theorem 4.7. There exists a 2DFA N_{B_m} deciding B_m with $p = p_1^{k_1} + ... + p_n^{k_n}$ states.

Proof. The working principle of N_{B_m} is simple: It has n groups of states and each group i has $p_i^{k_i}$ states, connected to form a cycle of states. First, it scans the input a^w from one side to the other and decides whether w is a multiple of $p_1^{k_1}$ with the help of its first group of $p_1^{k_1}$ periodic states. It also does this in a sweeping manner, meaning it turns its head only on the endmarkers. If w is actually a multiple of $p_1^{k_1}$, then N_{B_m} repeats this procedure for the second group of $p_2^{k_2}$ states, this time its head moving from right to left. That means in i^{th} traversal, it uses the i^{th} group of $p_i^{k_i}$ periodic states to check whether the length of a^w is a multiple of $p_i^{k_i}$. If w is indeed a multiple of m, then at the end of n^{th} traversal, it accepts; otherwise it does not accept. □

Now, we will prove that the smallest 2NFA deciding B_m cannot have fewer than p states, as the 2DFA we described just above does.

Theorem 4.8. Every 2NFA deciding B_m needs at least $p = p_1^{k_1} + ... + p_n^{k_n}$ states.

Proof. Suppose a p'-state 2NFA N' decides B_m where $p' < p$. Now, take an accepting computation c on the word a^m. Suppose also that c consists of some number of traversals and u-turns where the traversals use a total of $p'' \leq p'$ states. We investigate two scenarios:

- Disjoint traversals: Suppose that in c, we have a total of r traversals and every traversal t_i hosts a set of states N_i (i.e. the set of all states that can be encountered

in t_i). We assume that every such N_i is disjoint from one another. Say the size of each N_i is $|N_i| = n_i$.

In some traversal t_i where $1 \leq i \leq r$, since $m > p > n_i$, there must exist at least one state that is visited repeatedly first in the x^{th} and then in the $x + j^{th}$ positions of a^m where $x > 0$ and $j > 0$. Now, we choose from such repetitions the shortest possible one and call it j_i. We do that for every t_i, and so we have a set of integers $\{j_1, j_2, ..., j_r\}$.

Using an argument similar to the one used in theorem 4.2, we can pump the whole computation by the length $J = \text{lcm}(j_1, j_2, ..., j_r)$. Hence, the word a^{m+J} can be accepted by N' by an accepting computation c' which imitates c: all traversals are replaced by J-pumped traversals and u-turns, which may also exist in c, remain the same. This means that $J = km$ for some positive integer k. Now, say the prime factorization of J is $J = \prod_{i=1}^{n'} u_i^{v_i}$ where $u_i^{v_i}$ are prime factors of J. By using simple facts about primes and least common multiples, one can see that $\sum_{i=1}^{r} j_i \geq \sum_{i=1}^{n'} u_i^{v_i}$ and $\sum_{i=1}^{n'} u_i^{v_i} \geq \sum_{i=1}^{n} p_i^{k_i}$ and that gives the following:

$$p' \geq p'' = \sum_{i=1}^{r} n_i \geq \sum_{i=1}^{r} j_i \geq \sum_{i=1}^{n'} u_i^{v_i} \geq \sum_{i=1}^{n} p_i^{k_i} = p \tag{4.4}$$

Henceforth, we arrive at a contradiction, and p' cannot be smaller than p.

- Not disjoint traversals: Suppose that in c, we have r traversals t_is where each one host a set of states that may or may not be disjoint from every set that is hosted by other traversals. Now we will continue in a way similar to the previous scenario.

  Take two traversals t_g and t_h from c such that t_g has a repeating state q_g and t_h has a repeating state q_h with the following conditions:

  (i) q_g is visited repeatedly first in the x_g^{th} and then in the $x_g + j_g{}^{th}$ positions of a^m where $x_g > 0$ and $j_g > 0$,

  (ii) q_h is visited repeatedly first in the x_h^{th} and then in the $x_h + j_h{}^{th}$ positions of a^m where $x_h > 0$ and $j_h > 0$,

  (iii) j_g and j_h are the shortest state repetitions that can be seen in t_g and t_h, respectively.

Say each repetition in t_g and t_h hosts a set of states N_g and N_h, respectively. There are two possibilities: $N_g \cap N_h = \emptyset$ or $N_g \cap N_h \neq \emptyset$. First case is similar to the one we talked in the case of disjoint traversals and can be handled in a similar way by taking both of them separately into the pumping argument. Second case is more challenging and we analyze it now.

Take a state $q \in N_g \cap N_h$. We know that j_g and j_h are the shortest state repetitions and q is seen in both traversal parts. One can see that q also can be repeated in a pumped traversal as q_g and q_h can be repeated in their respective pumped traversals (the transition function allows that). Thus, both traversals can be pumped either by the length j_g or by the length j_h, consequently using the set of states N_g or N_h. We take the smallest of N_g and N_h as the set representative of t_g and t_h.

For a more generalized argument, suppose there exist a total of r traversals in c. Since they can be non-disjoint, one can find a total of $r' \leq r$ set representatives for all traversals such that any two set representatives are disjoint and any two sets of represented traversals are also disjoint. Since any set representative N_i has an associated original traversal t_i and a repetition length j_i, we can pump t_i and all the other traversals N_i represent with the length $j_i = |N_i|$. We do that for every N_i, and so we have a set of integers $\{j_1, j_2, ..., j_{r'}\}$ that is associated with the set of representatives $\{N_1, N_2, ..., N_{r'}\}$. We use this set to pump a^m.

Using an argument similar to the one used for disjoint traversals, we can pump the whole computation by the length $J = \text{lcm}(j_1, j_2, ..., j_{r'})$. Hence, the word a^{m+J} can be accepted by N' by an accepting computation c' which imitates c: all traversals are replaced by J-pumped traversals and u-turns, which may also exist in c, remain the same. This means that $J = km$ for some positive integer k. Now, say the prime factorization of J is $J = \prod_{i=1}^{n'} u_i^{v_i}$ where $u_i^{v_i}$ are prime factors of J. By using simple facts about primes and least common multiples, one can see that $\sum_{i=1}^{r} j_i \geq \sum_{i=1}^{n'} u_i^{v_i}$ and $\sum_{i=1}^{n'} u_i^{v_i} \geq \sum_{i=1}^{n} p_i^{k_i}$ and that gives the following:

$$p' \geq p'' \geq \sum_{i=1}^{r'} |N_i| = \sum_{i=1}^{r'} j_i \geq \sum_{i=1}^{n'} u_i^{v_i} \geq \sum_{i=1}^{n} p_i^{k_i} = p \tag{4.5}$$

Therefore, we arrive at a contradiction, and p' cannot be smaller than p.

Hence, the theorem is proved. □

This result is compatible with the result given in Mereghetti and Pighizzini's paper [26]. It is worth noting that if m is a prime power, say $m = p^k$ for some prime number p, then 2NFA cannot outperform DFA in deciding B_m with fewer number of states. Now, as for the case of A_m, we can turn our direction to AFA deciding B_m:

Theorem 4.9. The minimal AFA that decides B_m has $\lceil \log m \rceil$ states.

Proof. Using theorem 3.13, we know that every n-state minimal DFA has an equivalent minimal $\lceil \log n \rceil$-state AFA. D_{B_m} has m states, thus the minimal AFA that decides B_m has $\lceil \log m \rceil$ states. Hence, the theorem is proved. □

We do not know if 2AFA can outperform AFA in deciding B_m. Now, we will turn to PFA and show that PFA can perform (statewise) better than DFA:

Theorem 4.10. There exists a PFA that decides B_m with $p + 2$ states where $p = p_1^{k_1} + ... + p_n^{k_n}$ and its error margin is equal to $\frac{2n-1}{4n}$.

Proof. Here, we will use a slightly modified version of the notion used in 2DFA case: Instead of traversing the input for n times sequentially with n different groups of states as the 2DFA N_{B_m} does, our PFA does it parallelly with n groups of states. Every group i has $p_i^{k_i}$ states that form a cycle where a cycle is disjoint from other cycles. On each cycle i, the starting state is also the final state, hence the i^th cycle can be seen as a ${p_i}^{k_i}$-state 1DFA which accepts strings of length $c{p_i}^{k_i}$, $c \in \mathbb{N}$. Also, the starting distribution is different than usual, meaning that instead of using 1 state as the starting state with probability 1 at the beginning, we use the starting states of every group as one of n starting states, each having the probability $\frac{1}{n}$ at the beginnning (i.e. instead of using $(1 \ 0 \ 0 \ ... \ 0)^T$ vector, we use $(\frac{1}{n} \ 0 \ ... \ 0 \ \frac{1}{n} \ 0 \ ... \ 0 \ ... \ ... \ \frac{1}{n} \ 0 \ ... \ 0)^T$ vector at the start).

During the computation of some word a^r, our PFA does the following: In some step, for every group i, we are in one of the states in the i^{th} group with $1/n$ probability, trying to figure out if r is a multiple of ${p_i}^{k_i}$. Thus, it seems like we force every group to find whether r is a multiple of $m = p_1^{k_1}...p_n^{k_n}$. If indeed r is a multiple of m, we are in the final state of every group with $\frac{1}{n}$ probability (adding up to 1), hence a^r is accepted by our PFA with probability 1. In other cases, it is certain that our PFA accepts a^r with at most $1 - \frac{1}{n}$ probability.

Now, we will modify the PFA described above. To do that, we add two additional starting states, one accepting and one rejecting, both are cycles of length one and disjoint from every other group. Then, we change the starting probability distribution vector to

$$(\frac{1}{2n} \;\; 0 \;\; ... \;\; 0 \;\; \frac{1}{2n} \;\; 0 \;\; ... \;\; 0 \;\; ... \;\; ... \;\; \frac{1}{2n} \;\; 0 \;\; ... \;\; 0 \;\; \frac{1}{4n} \;\; \frac{1}{2} - \frac{1}{4n})^T$$

so every starting state of the old groups now has $\frac{1}{2n}$ starting probability; moreover, the newly added accepting starting state and its rejecting counterpart have $\frac{1}{4n}$ and $\frac{1}{2} - \frac{1}{4n}$ starting probability, respectively. We do not alter any other computational property of our PFA. By doing those modifications, we change the acceptance probability of a given string a^r as the following:

- if r is a multiple of m, then $p(r) = n\frac{1}{2n} + \frac{1}{4n} = \frac{2n+1}{4n}$
- if r is not a multiple of m, then $p(r) \leq (n-1)\frac{1}{2n} + \frac{1}{4n} = \frac{2n-1}{4n}$

Hence, we can set the error margin as $1 - \frac{2n+1}{4n} = \frac{2n-1}{4n}$. Notice that the margin is equal to $\frac{1}{2} - O(n^{-1})$ and we cannot fix an error margin that is same for every n, using only $p+2$ states. Also, this result is compatible with the result given in Mereghetti, Palano, and Pighizzini's paper [12] in which the machine of concern is a different version of PFA (i.e. PFA with isolated cut-point). □

One can see that we can fix n and fix the error margin. For example, consider a special case of m where m is a multiple of two prime factors $p_1^{k_1}$ and $p_2^{k_2}$ which are approximately equal. Using the PFA described just above, one can create a PFA with $p + 2 = p_1^{k_1} + p_2^{k_2} + 2 \approxeq \sqrt{m} + \sqrt{m} + 2 = O(\sqrt{m})$ states and error margin $\epsilon = \frac{3}{8}$.

It is worth noting that if $m = p^k$ for some prime number p, then the aforementioned PFA model (i.e. grouping states in cycles and using them parallelly) cannot outperform DFA in deciding B_m with fewer number of states. Additionally, if m is a prime number, then any bounded-error PFA cannot decide B_m with number of states fewer than m, stated in Ambainis and Freivalds' paper [10].

Now, we will turn to QFA and answer the following question:

"For any $\epsilon > 0$, does there exist a bounded-error QFA that decides B_m with $O(\log m)$ states and ϵ error margin where m is a prime number?"

Apparently, there exists (see [10]). To carefully explain how a bounded-error QFA does that, we will use the following:

Theorem 4.11. For every prime number m, there exists a 2-state QFA that accepts the members of B_m with 1 probability and non-members of B_m with at most $\cos^2 \frac{\pi}{m}$ probability.

Proof. Now, we formally give a QFA P deciding B_m: $P = (\{q_1, q_2\}, \{a\}, U, q_1, \{q_1\})$ where U is the orthogonal matrix:

$$U = \begin{bmatrix} \cos\frac{\pi}{m} & -\sin\frac{\pi}{m} \\ \sin\frac{\pi}{m} & \cos\frac{\pi}{m} \end{bmatrix}$$

Given a string $w = a^r$, when P reads the symbol a on i^{th} step, it does a simple operation $U\rho_{w,i-1}U^\dagger$ to compute $\rho_{w,i}$, using its only matrix operator U and the previous computation $\rho_{w,i}$. Geometrically speaking, U is a transformation that rotates a point

on the unit circle in the Euclidean plane by the angle $\frac{\pi}{m}$ in counter-clockwise direction. What it does in our case is more or less the same thing since at every step it changes the probability of being on a state by some amount that depends on the angle of rotation $\frac{\pi}{m}$ and the previous angle. As an example, in the i^{th} step, the computation is equal to

$$\rho_{w,i} = \begin{bmatrix} \cos^2 \frac{\pi i}{m} & ... \\ ... & \sin^2 \frac{\pi i}{m} \end{bmatrix}$$

Thus, the probability that P accepts w is $p(w) = \cos^2 \frac{\pi r}{m}$. If r is a multiple of m, $p(w) = 1$; if not, then $p(w) \leq \cos^2 \frac{\pi}{m}$. Hence, we prove the theorem. Notice that P is not a bounded-error QFA and its error margin depends on m. □

Now, using P, we will build a bounded-error QFA, and prove the following theorem.

Theorem 4.12. For any $\epsilon > 0$, there exists a $O(\log m)$-state bounded-error QFA that decides B_m with error margin ϵ where m is a prime number [10].

Proof. Notice that instead of using the angle $\frac{\pi}{m}$ for rotations, our PFA P (given in the previous proof) can use the angle $\frac{j\pi}{m}$ where j is any non-negative integer smaller than m, and can still accept $w \in B_m$ with $p(w) = 1$ and $w \notin B_m$ with $p(w) \leq \cos^2 \frac{\pi}{m}$ (it is essentially because every such j and m are coprime). Thus, denoting P_j as our modified PFA with rotation angle $\frac{j\pi}{m}$, $P_j = (\{q_{j,1}, q_{j,2}\}, \{a\}, U_j, q_{j,1}, \{q_{j,1}\})$ having U_j such that

$$U_j = \begin{bmatrix} \cos \frac{j\pi}{m} & -\sin \frac{j\pi}{m} \\ \sin \frac{j\pi}{m} & \cos \frac{j\pi}{m} \end{bmatrix}$$

also decides B_m. In fact, one can see that for $i, j \in \{0, 1, 2, ..., m-1\}$, the lists $[\cos^2 0, \cos^2 \frac{i\pi}{m}, \cos^2 \frac{2i\pi}{m}, ..., \cos^2 \frac{(m-1)i\pi}{m}]$ and $[\cos^2 0, \cos^2 \frac{j\pi}{m}, \cos^2 \frac{2j\pi}{m}, ..., \cos^2 \frac{(m-1)j\pi}{m}]$ have the same elements, just in different order, if $i \neq j$. These lists also can represent the associated acceptance probability of list of words $[a^0, a^1, a^2, ..., a^{m-1}]$ by P_i and P_j,

respectively. We will utilize this different order notion by using the following lemma:

Lemma 4.13. There exists a set $J = \{j_1, j_2, ..., j_k\} \subset \{0, 1, ..., m-1\}$ of $k = O(\frac{1}{\epsilon} \log m)$ numbers such that for every $j \in J$, the elementwise sum of lists $L_j = [\cos^2 0, \cos^2 \frac{j\pi}{m}, \cos^2 \frac{2j\pi}{m}, ..., \cos^2 \frac{(m-1)j\pi}{m}]$ is equal to

$$L = \sum_{j=1}^{k} L_j = [l_0, l_1, l_2, ..., l_{m-1}] \tag{4.6}$$

where $l_0 = k \cos^2 0 = k$ and $l_y \leq \epsilon k$ for any $1 \leq y \leq m-1$ (elementwise sum of two equal-size lists $L_1 = [a_1, a_2, ..., a_k]$ and $L_2 = [b_1, b_2, ..., b_k]$ is another list $L_3 = [a_1 + b_1, a_2 + b_2, ..., a_k + b_k]$).

In Ambainis and Nahimovs' paper [27] (and similarly in the paper [10]), this lemma is proven. By the way, this is an existence proof; how to choose such a set is not given explicitly, but we know that such a set exists. Nevertheless, we can use the members of J for the entries in U_js of our QFA P_js, and by that, we can build a bigger QFA, say P' with a unitary matrix U', that accepts the members of B_m with probability 1 and non-members with probability smaller than or equal to ϵ.

For the sake of clarity and simplicity, we will slightly modify our QFA's working mechanism on B_m before precisely describing P'. For this case, we will do a "precomputation" and then read the input a^r. We will represent this as reading the word $\#a^r$. The final measurement of observation is still done at the end.

Precisely, $P' = (Q'', \{a, \#\}, \{U', U_\#\}, q_{1,1}, F'')$ such that $F'' = \{q_{1,1}, q_{2,1}, ..., q_{k,1}\}$, $Q = \{q_{1,1}, q_{1,2}, q_{2,1}, q_{2,2}, ..., q_{k,1}, q_{k,2}\}$, and U' and $U_\#$ are given as:

$$U' = \begin{bmatrix} \cos\frac{j_1\pi}{m} & -\sin\frac{j_1\pi}{m} & 0 & ... & ... & ... & ... & 0 \\ \sin\frac{j_1\pi}{m} & \cos\frac{j_1\pi}{m} & 0 & ... & ... & ... & ... & ... \\ 0 & 0 & \cos\frac{j_2\pi}{m} & -\sin\frac{j_2\pi}{m} & 0 & ... & ... & ... \\ 0 & 0 & \sin\frac{j_2\pi}{m} & \cos\frac{j_2\pi}{m} & 0 & ... & ... & ... \\ ... & ... & 0 & 0 & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & 0 & 0 \\ ... & ... & ... & ... & ... & 0 & \cos\frac{j_k\pi}{m} & -\sin\frac{j_k\pi}{m} \\ 0 & ... & ... & ... & ... & 0 & \sin\frac{j_k\pi}{m} & \cos\frac{j_k\pi}{m} \end{bmatrix}$$

$$U_{\#} = \begin{bmatrix} \frac{1}{\sqrt{k}} & 0 & \frac{1}{\sqrt{k}} & 0 & ... & ... & \frac{1}{\sqrt{k}} & 0 \\ 0 & 1 & 0 & 0 & ... & ... & ... & ... \\ \frac{1}{\sqrt{k}} & 0 & \frac{1}{\sqrt{k}} & 0 & ... & ... & \frac{1}{\sqrt{k}} & 0 \\ 0 & 0 & 0 & 1 & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & ... & ... \\ \frac{1}{\sqrt{k}} & 0 & \frac{1}{\sqrt{k}} & 0 & ... & ... & \frac{1}{\sqrt{k}} & 0 \\ ... & ... & ... & ... & ... & ... & 0 & 1 \end{bmatrix}$$

Now, we will explain what our QFA does, starting with describing the precomputation matrix $U_{\#}$. At the beginning, it uses the orthogonal transformation matrix $U_{\#}$ to change the initial density matrix, say M_0, into a matrix M_k (by $U_{\#}M_0U_{\#}^{\dagger} = M_k$) and the odd-numbered diagonal entries $(2j-1, 2j-1)$ of M_k are equal to $\frac{1}{k}$ (which are actually associated with the states $q_{j,1}$). This precomputation gives sort of an equal "starting" probability $\frac{1}{k}$ to every $q_{j,1}$, the de facto starting states of U_js. Then, it reads the main input a^r. At the end, we accept the input with probability $p(w)$ which is equal to the sum of odd-numbered diagonal entries of our QFA's final computation, given by the definition of acceptance of QFA and the fact that our final states are every $q_{j,1}$. Now, we will explain the main reading part of P'.

Graphically speaking, we can represent P_j as a simple clock having only one hand that rotates in counter-clockwise direction with the angle $\frac{j\pi}{m}$ every time it sees the symbol a. Thus, concerning reading the input, what P' does is actually very simple: It simulates every clock P_j simultaneously, using U' (which unsurprisingly consists of every U_j, diagonally binded to one another). After reading an input a^t, the clock P_j's hand makes an angle $\theta = \frac{jt\pi}{m} = \frac{t_j\pi}{m}$ with the x-coordinate where $jt \equiv t_j \pmod{m}$ and $0 \leq t_j < m$. This means that the entry in the $2j-1^{th}$ diagonal of our final computation becomes $\frac{\cos^2 \frac{t_j\pi}{m}}{k}$. If t is a multiple of m, then the total sum of such entries is 1, implying that a^t is accepted by $p(a^t) = 1$ probability. If not, using the knowledge about the list L stated in 4.13, we know that the total sum cannot be more than $\frac{\epsilon k}{k} = \epsilon$, implying that a^t is accepted by at most $p(a^t) = \epsilon$ probability. So actually, we use every entry of L to calculate as the acceptance probability of a given input. Therefore, with $2k = O(\log m)$ states, a QFA can decide B_m. This completes the proof.

As a note, one can see that the QFA that is constructed in this proof is actually a OQFA. □

4.3. The Language Class C_m and Its Deciding Automata

In this section, we will introduce a binary regular language shortly noted by C_m. C_m, which is proper subset of $\{0,1\}^*$, is the language of binary strings that has a "0" as its m^{th} bit from the end. We can start by stating the following simple theorem:

Theorem 4.14. Considering state-wise efficiency, NFA are logarithmically more concise than DFA while deciding C_m.

Proof. As we showed in the proof of theorem 3.2, we know that a NFA can recognize C_m with only $m+1$ states and a DFA recognizing C_m needs at least 2^m states. Hence, the proof is done. □

Now, we will prove that 2DFA can decide C_m with $m+2$ states.

Theorem 4.15. Considering state-wise efficiency, 2DFA are logarithmically more concise than DFA while deciding C_m.

Proof. The trick here is to see that 2DFA can decide the reverse language of C_m with $m+2$ states. Say a 2DFA D_{C_m} decides C_m. The computation procedure of 2DFA on the word w is simple. Starting in the state q_0, it scans the input from left to right while staying in the same state. On the right endmarker, it jumps into the state q_1 and turns its head left. Then, it scans the input from right to left while going into the next state at every jump, or precisely speaking, being on the i^{th} symbol from the end of w, D_{C_m} goes from the state q_i to q_{i+1} and moves left. On the m^{th} symbol from the end of w, being in the state q_m, it checks whether the symbol is 0 or not. If it is, it turns its head right again and goes into the state q_{m+1}. It scans the m-symbol long part again from left to right while staying in q_{m+1}, goes right past the right endmarker and accepts. In any other case, it halts and rejects. The formal definition of D_{C_m} is: $D_{C_m} = (\{q_0, ..., q_{m+1}\}, \{0,1\}_{\{\$,\#\}}, \delta, q_0, \{q_{m+1}\})$ where

- $\delta(q_0, \$) = \delta(q_0, 0) = \delta(q_0, 1) = (q_0, r)$
- $\delta(q_0, \#) = (q_1, l)$
- $\delta(q_i, 0) = \delta(q_i, 1) = (q_{i+1}, l)$ for $1 \leq i \leq m-1$
- $\delta(q_m, 0) = \delta(q_{m+1}, 0) = \delta(q_{m+1}, 1) = \delta(q_{m+1}, \#) = (q_{m+1}, r)$

One quick addition is that if we allow 2DFA to begin on the right endmarker (instead of left endmarker), we can save one more state. The proof is done. □

Now, we will check the number of states necessary for 2NFA to decide C_m:

Theorem 4.16. At least $m-1$ states are necessary for a 2NFA to decide C_m.

Proof. Suppose an m'-state 2NFA N' decides C_m where $m' < m-1$. Now, take an accepting computation c on the word $w = 01^{m-1}$ where $|w| = m$. By definition of acceptance, this c should start in the left endmarker and visit the right endmarker.

That means there exists at least one traversal t in c and there may exist u-turns. We will start with the consideration of traversals and then analyze possible u-turns:

- For a traversal t, since $m' < m-1$, there exists at least one state that is visited repeatedly first in the x^{th} and then in the $x+j^{th}$ positions of the input suffix 1^{m-1}. Thus, if we pump w by 1^j from it right end, we can show that there exists another traversal t' imitating a j-pumped t, in a similar way used to prove the theorem 4.2 (i.e. t' starts and ends on the same states as t does).
- c can also have u-turns beginning from the right endmarker and ending on the right endmarker. Suppose such a u-turn u includes the reading on the input suffix 1^{m-1}. Since $m-1 > m'$, there exists at least one state that is visited repeatedly first in the $x+j^{th}$ and then in the x^{th} positions of the input suffix 1^{m-1}. Thus, we can pump u by adding j units of 1s to its rightend (i.e. shifting its leftmost turning position to the left by j on the new extended input).

Say there exist such j_is of c, namely $j_1, j_2, ..., j_k$, on all traversals and all special "from-right-to-right" u-turns, which has the characteristics described above, namely $t_1, t_2, ..., t_l$, and $u_1, u_2, ..., u_k$ then, we can pump the whole computation with the length $J = j_1 * j_2 * ... * j_l * u_1 * u_2 * ... * u_k$. Hence, the word 01^{m-1+J} can be accepted by N' with an accepting computation c' which imitates c: traversals are replaced by pumped traversals and those special u-turns are replaced by pumped u-turns (other u-turns remain the same). Since 01^{m-1+J} is not a member of C_m, N' should not accept it, giving a contradiction. Hence, at least $m-1$ states are necessary for a 2NFA to decide C_m and the proof is done. □

Now, we turn our direction to AFA case:

Theorem 4.17. Considering state-wise efficiency, AFA are double-logarithmically more concise than DFA while deciding C_m.

Proof. We will mainly restate theorem 3.11. We know that a DFA needs at least 2^m states to decide C_m. Also, we can easily see that the reverse language of C_m, namely

${C_m}^R$, can be decided by a DFA with $m+2$ states (i.e. ${C_m}^R$ is the language of binary strings that has a "0" as its m^{th} bit from the start, so a DFA should only check the m^{th} bit by using a counter that can count upto $m+2$). Using the theorem 3.10, we infer that the reverse language of ${C_m}^R$, which is actually ${{C_m}^R}^R = C_m$, can be decided by an AFA with $\log\lceil m+2\rceil$ states. Thus, the theorem is proved. □

We do not know if 2AFA can outperform AFA in deciding C_m. Now, we turn our direction to PFA and QFA:

Theorem 4.18. There exists a PFA that decides C_m with $m+2$ states and its error margin is $\epsilon = \frac{1}{2} - O(\frac{1}{m})$ [28].

Proof. Our proof consists of a slightly modified version of a PFA given in Rasscevskis' paper [28]. First, we build a PFA, and then we show why it decides C_m with given ϵ.

Suppose we have $m+2$ states $\{q_A, q_0, q_1, q_2, ..., q_m\}$ where q_A and q_m are the only final states. Furthermore, q_A and q_0 are the only starting states and the starting distribution of states is $(\frac{1}{2}-\gamma \;\; \frac{1}{2}+\gamma \;\; 0 \;\; 0 \;\; 0 \;\; ... \;\; 0)^T$ for some γ that will be precisely defined later in the proof. Now, we will describe the transition between states:

- On any symbol, q_A has a transition to itself with probability 1.
- On symbol 0, q_0 has a transition to itself with probability x and a transition to q_1 with probability $1-x$.
- On symbol 1, q_0 has a transition to itself with probability 1.
- On any symbol, q_i has a transition to q_{i+1} with probability 1 for any $i < m$.
- On any symbol, q_m has a transition to q_0 with probability 1.

Clearly, our PFA computes in the following way: It starts on q_A with $\frac{1}{2}-\gamma$ probability and q_0 with $\frac{1}{2}+\gamma$ probability. If q_A is chosen, it stays on q_A. If q_0 is chosen, then it shows a cyclic behaviour with some probability on inputs having more than one 1; on other inputs, it stays on q_0. One can see that this cyclic behaviour of length $m+1$ is a result of the transition function defined for states $\{q_0, q_1, q_2, ..., q_m\}$.

Now, we will prove the following three properties of this PFA:

(i) For every $w \notin C_m$, the accepting probability of w is $p(w) = \frac{1}{2} - \gamma$.

(ii) The computation on any word w ends in q_0 with at least $(\frac{1}{2} + \gamma)x^m$ probability.

(iii) For every $w \in C_m$, $p(w) \geq \frac{1}{2} - \gamma + (\frac{1}{2} + \gamma)x^m(1 - x)$.

Let's prove all these three properties:

(i) Suppose the input $w = u1v$ is given where $u \in \{0,1\}^x$ and $|v| = m - 1$. That means with some probability $d > 0$, the computation on w ends in state q_m (the only accepting states are q_A and q_m). Thus, the computation on its prefix u ends in q_0 with d. However, reading the next symbol 1 should result in staying on q_0, according to the transition rules of our PFA. Therefore, reading the remaining part v finishes on a state other than q_m since $|v| = m - 1$. Hence, $d = 0$, giving a contradiction. Thus, $p(w) = \frac{1}{2} - \gamma$.

(ii) First, we should check the case where $|w| \leq m$. Since its length is smaller than the length of our $m+1$ state-long cycle, the computation on w can end in q_0 only if it starts on q_0 and stays on q_0 at every step. Thus, the computation on w ends in q_0 with probability $(\frac{1}{2} - \gamma)^{|w|}$, which is never less than $(\frac{1}{2} - \gamma)^m$ when $|w| \leq m$. Now, we check the second case that $|w| > m$. In this case, the computation can either start and stay in q_0, or it can complete our cycle (perhaps more than one, if $|w|$ is big enough) and then return to q_0. Say d is the probability of ending on q_0 after reading w. Clearly, to hit q_0 at the end, the computation should hit one of the states in $\{q_0, q_1, q_2, ..., q_m\}$ m steps before the end. Say P_i denotes the probability that the computation of w hits the state q_i m symbols before the end. Furthermore, for any integer i where $1 \leq i \leq m$, if the computation ends in q_0 and hits q_i m steps before the end, for the final m steps, it should have traversed the last $m - i$ states of the cycle upto q_m, reached q_0, and have stayed on q_0 for the final $i - 1$ steps. This situation can happen with probability $P_i x^{i-1}$. p cannot be less than the addition of all such probabilities $P_i x^{i-1}$ and the probability $P_0 x^m$ that the computation hits the state q_0 m symbols before the end and still ends in q_0. We also now that $\sum_{i=0}^{m} P_i = \frac{1}{2} + \gamma$ and $x^{i-1} \geq x^m$ for any i and x such that

$1 \leq i \leq m$ and $0 \leq x \leq 1$. Therefore, we have the following:

$$d \geq P_0 x^m + \sum_{i=1}^{m} P_i x^{i-1} \geq P_0 x^m + \sum_{i=1}^{m} P_i x^m = x^m \sum_{i=0}^{m} P_i = (\frac{1}{2} + \gamma) x^m \tag{4.7}$$

Henceforth, the computation on any word w ends in q_0 with at least $(\frac{1}{2} + \gamma)x^m$ probability.

(iii) For every $w \in C_m$, we know that $p(w) = \frac{1}{2} - \gamma + d'$ where d' denotes the probability that the computation on w ends in the state q_m. d' depends on d and $1-x$, which denote the probability of hitting q_0 m symbols before the end of the computation and the probability of going from q_0 to q_1, respectively. One can see that $d' = d(1-x)$. Using this with the second property results in the following:

$$p(w) = \frac{1}{2} - \gamma + d(1-x) \geq \frac{1}{2} - \gamma + (\frac{1}{2} + \gamma)x^m(1-x) \tag{4.8}$$

Hence, for every $w \in C_m$, $p(w) \geq \frac{1}{2} - \gamma + (\frac{1}{2} + \gamma)x^m(1-x)$.

The only thing remaining is to fix the error margin ϵ by benefiting from the third property and the necessary condition that for every $w \in C_m$, $p(w) \geq \frac{1}{2} + \gamma$. Thus, one must satisfy the following:

$$\begin{aligned} \frac{1}{2} - \gamma + (\frac{1}{2} + \gamma)x^m(1-x) &\geq \frac{1}{2} + \gamma \\ (\frac{1+2\gamma}{2})x^m(1-x) &\geq 2\gamma \\ x^m(1-x) &\geq \frac{4\gamma}{1+2\gamma} \end{aligned} \tag{4.9}$$

It is satisfactory that $x^m(1-x) \geq 4\gamma$. By fixing m and using derivation, one can find the optimal value for x as $x = \frac{m}{m+1}$, implying $(\frac{m}{m+1})^m(1-\frac{m}{m+1}) \geq 4\gamma$. Thus, $\gamma = O(\frac{1}{m})$, making $\epsilon = \frac{1}{2} - O(\frac{1}{m})$, and completing the proof. □

Theorem 4.19. There exists a QFA that decides C_m with $m+2$ states and its error margin is $\epsilon = \frac{1}{2} - O(\frac{1}{m})$.

Proof. The theorem actually says that we can construct a QFA that decides C_m by modifying the PFA we give in the theorem 4.18, without changing the error margin or the number of states. It mainly comes from the following theorem, stated in Say and Yakaryilmaz's paper [20].

Theorem 4.20. For a given n-state PFA M, there exists an n-state QFA N such that the acceptance probability of any string w by M is equal to that of N: $p_M(w) = p_N(w)$.

Proof. The proof is simple. We will give a QFA N that correctly simulates a PFA M.

Say $M = (\{q_1, q_2, ..., q_n\}, \Sigma, \{E_\sigma\}_{\sigma\in\Sigma}, \delta, q_1, F)$ where every E_σ is a transition matrix defined for $\sigma \in \Sigma$. Then, we construct N as $N = (\{q_1, q_2, ..., q_n\}, \Sigma, \{\mathcal{E}_\sigma\}_{\sigma\in\Sigma}, q_1, F)$ where for $\sigma \in \Sigma$, $\mathcal{E}_\sigma = \{E'_{\sigma_j} \mid j \in [1, 2, ..., n]\}$.

Suppose some word w is given to M. Let v_1 be probability disribution vector associated with w by M:

$$v_1 = \begin{bmatrix} p_1 \\ p_2 \\ ... \\ ... \\ p_n \end{bmatrix}$$

where $\sum_{i=1}^{n} p_i = 1$ (of course). Then, N can simulate M's computation on w by representing v_1 with V, the probability disribution matrix associated with w by N as:

$$V = \begin{bmatrix} p_1 & ... & ... & ... & ... \\ ... & p_2 & ... & ... & ... \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & p_n \end{bmatrix}.$$

One can see that $p_M(w) = p_N(w)$. Furthermore, take E_b, the transition matrix of M which is associated with the symbol $b \in \Sigma$, which is defined as:

$$E_b = \begin{bmatrix} a_{1,1} & a_{1,2} & ... & ... & a_{1,n} \\ a_{2,1} & a_{2,2} & ... & ... & a_{2,n} \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ a_{n,1} & a_{n,2} & ... & ... & a_{n,n} \end{bmatrix}.$$

The computation of M on wb results in the vector $v_1' = E_b v_1$. Now, take E_{b_j}', one of the n operators associated with the symbol $b \in \Sigma$. j^{th} column of E_{b_j}' is (and should be) equal to:

$$\begin{bmatrix} \sqrt{a_{1,j}} \\ \sqrt{a_{2,j}} \\ ... \\ ... \\ \sqrt{a_{n,j}} \end{bmatrix}$$

and all other entries are zero. The computation of N on wb results in the matrix $V' = \sum_{j=1}^{n} E_{b_j}' V E_{b_j}'^{\dagger}$. One can see that $p_M(wb) = p_N(wb)$. Thus, N is a well-formed QFA that correctly simulates M. □

Thus, we can use the PFA described in theorem 4.18 to create a QFA that decides C_m with $m + 2$ states and error margin $\epsilon = \frac{1}{2} - O(\frac{1}{m})$. Hence, the proof is done. □

It is worth mentioning that we do not know whether for any $\epsilon > 0$, there exists a PFA or QFA deciding C_m with $O(m)$ states with bounded error. This concludes the fourth chapter.

5. UNARY FINITE PERIODIC FORM

5.1. Preliminaries

In the beginning of this part, we give basic information about some interrelated topics in Automata Theory, Number Theory, and Descriptional Complexity Theory. We start by recalling unary finite automata and unary regular languages:

Definition 5.1. A *unary DFA*, shortly *UDFA*, is a DFA working on an alphabet containing only 1 symbol. For instance, a UDFA D_3, that is defined as:

$$D_3 = (\{q_1, q_2, q_3\}, \{a\}, \{\delta(q_1, a) = q_2, \delta(q_2, a) = q_3, \delta(q_3, a) = q_1\}, q_1, \{q_1\})$$

accepts strings of the form a^{3m} for $m \in \mathbb{N}$.

Definition 5.2. A *unary regular language*, shortly *URL*, is a regular language decided by a UDFA. For instance,

$$L_{D_3} = \{a^{3m} \mid m \in \mathbb{N}\}$$

is decided by the example UDFA D_3 given above. A URL can also be seen as a special kind of encoding of a set of natural numbers in the unary numeral system. Furthermore, URLs can have finite or infinite number of members, as regular languages do.

Definition 5.3. A set S of nonnegative integers is called *ultimately periodic* if there exist constants $\mu \geq 0$ and $\lambda > 0$ such that every $k \geq \mu$ belongs to S if and only if $k + \lambda$ belongs to S. μ and λ are called *transience* and *period*, respectively. Also, S is called *periodic* if $\mu = 0$. For example

$$S_{5,7} = \{k \mid k = 3 \text{ or } k = 4 \text{ or } k = 5 + 7m \text{ where } m \in \mathbb{N}\}$$

is ultimately periodic and it is a set of numbers whose members are exactly the positive integers 3, 4, and $k \equiv 5 \pmod 7$.

From these three simple definitions above, one can see that URLs actually form a good representation for the ultimately periodic sets. Thus, we can talk about a URL having the property of being periodic or not. Also, the states and transition function of a UDFA can capture the notions of transience and period. For instance,

$$L_{D_{1,2}} = \{a^m \mid m = 0 \text{ or } m = 2n + 1 \text{ where } n \in \mathbb{N}\},$$

which is decided by

$$D_{1,2} = (\{q_1, q_2, q_3\}, \{a\}, \{\delta(q_1, a) = q_2, \delta(q_2, a) = q_3, \delta(q_3, a) = q_2\}, q_1, \{q_1, q_2\}),$$

can be considered as a set with ultimately periodic behavior and has transience $\mu = 1$ and period $\lambda = 2$. Furthermore, the size of the set of states of a UDFA representing such an ultimately periodic set can be described as $|Q| = \mu + \lambda$. In fact, the states and transition functions of UDFA are usually represented in this way, meaning the visual representation of the transition function, or the transition graph, of a UDFA resembles a spoon having μ states linearly connected in its handle and λ states forming a circle in its bowl (see figure 5.1). As a final reminder, for UDFA, μ can be at least 0 (i.e. the regular language it decides represents a periodic set) and λ can be at least 1 (i.e. UDFA are complete), two properties which are similar to those of ultimately periodic sets.

Now, we state an observation about DFA in general. If we want to store all the information about a DFA D, we need to store the alphabet, states, transition function, the starting state, and accepting states of D. If we fix the alphabet size, then the main consideration should be on the size of data about states and their relations. Keeping it simple, we can say that storing information about an n-state DFA in binary should take $O(n \log n)$ bits: the states, transition function, accepting states, and starting state should take $n \log n$, $O(n \log n)$, at most $n \log n$, $\log n$ bits, respectively (alphabet size is fixed).

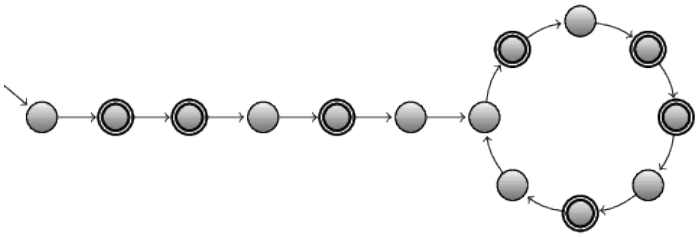

Figure 5.1. Transition graph of a 14-state UDFA with $\mu = 6$ and $\lambda = 8$.

Particularly for UDFA, where the alphabet size is 1, storing all data about the states can be quite a redundant thing to do. We claim that one can store only the information about accepting states (or accepted strings) in binary and still generate a form that can be used to describe and decide URLs just as UDFA do. For instance, the language $A_m = \{a^m\}$ is decided by an $m+2$-state minimal UDFA (proven in theorem 4.1), and the storage costs $O(m \log m)$ bits in the traditional way. Instead, if we only store the accepting state of L_m, that should take only $O(\log m)$ bits. Below, we will precisely define our new storage form that can act as a decider for URLs.

One quick addition, we assume that the states, final states, and the transition function table of UDFA are increasingly sorted (this assumption will help us when we construct our storage form). It means that for a UDFA $D = (\{q_0, q_1, ..., q_n\}, \{a\}, \delta, q_0, F)$, deciding some URL L, the points below are valid:

- $\delta(q_j, a) = q_{j+1}$ where $0 \leq j \leq n-1$,
- $q_i \in F$ if and only if $a^i \in L$.

5.2. Introducing Unary Finite Periodic Form

Definition 5.4. A *unary finite periodic form*, shortly *UFPF*, is a 5-tuple $(\alpha, \mu, \lambda, R, P)$:

(i) α is a symbol called the *alphabet symbol*,

(ii) μ is a non-negative integer called the *transience*,

(iii) λ is a positive integer called the *period*,

(iv) P is an increasingly sorted list of non-negative integers called the *periodic part*,

(v) R is an increasingly sorted list of non-negative integers called the *transient part* or *non-periodic part*,

where the largest member of R is smaller than μ and the largest member of P is smaller than λ.

Definition 5.5. For some URL L that is defined as:

$$L = \{a^x \mid x \in [\mu_1, \mu_2, ..., \mu_m]\} \cup \{a^{\mu+y} \mid (y \pmod{\lambda}) \in [\lambda_1, \lambda_2, ..., \lambda_n]\}$$

where

- for every non-negative integer μ_i, $\mu_i < \mu_{i+1}$ for $1 \leq i \leq m-1$ and $\mu_m < \mu$,
- for every non-negative integer λ_j, $\lambda_j < \lambda_{j+1}$ for $1 \leq j \leq n-1$ and $\lambda_n < \lambda$,

the associated unary finite periodic form U_L *describes* L where $U_L = (a, \mu, \lambda, R, P)$, $R = [\mu_1, \mu_2, ..., \mu_m]$, and $P = [\lambda_1, \lambda_2, ..., \lambda_n]$.

As an example, say a URL L_1 is defined as a union of two sets as

$$L_1 = \{a^1, a^2, a^4\} \cup \{a^6 a^x \mid x \geq 0 \text{ and } x \equiv i \pmod{4} \text{ where } i \in \{0, 1, 3\}\}.$$

Now, we can write the unary finite periodic form of L_1 as

$$U_{L_1} = (a, 110, 100, [001, 010, 100], [000, 001, 011])$$

in binary notation. Another example is for the finite URL $L_2 = \{\epsilon, b^5, b^6\}$ where

$$U_{L_2} = (b, 111, 001, [000, 101, 110], []).$$

The last example is for the periodic URL $L_3 = \{a^{9x} \mid x \in \mathbb{N}\}$ where

$$U_{L_3} = (a, 0, 1001, [], [0000]).$$

One additional note, every member is represented with the same number of bits. Strictly speaking, if the transience is μ and the period is λ for some UFPF U_L, then every member of R, or "transient member of U_L" or "non-periodic member of U_L", is represented with $\lceil \log \mu \rceil$ bits and every member of P, or "periodic member of U_L", is represented with $\lceil \log \lambda \rceil$ bits. Now, we will state that we can also use UFPFs to decide URLs.

Say a UFPF $U_L = (a, \mu, \lambda, [\mu_1, \mu_2, ..., \mu_m], [\lambda_1, \lambda_2, ..., \lambda_n])$ is given for the unary regular language $L = \{a^k \mid k = \mu_i$ or $k = \mu + c\lambda + \lambda_j$ where $1 \leq i \leq m$, $1 \leq j \leq n$, $c \in \mathbb{N}\}$. Suppose an integer l is given in binary form for the string a^l. U_L should decide whether a^l is in L or not by operating on l to find whether it includes l as a member. To do this, our UFPF searches its non-periodic and periodic parts. First, it checks whether $l < \mu$. If so, U_L searches the non-periodic part: If it finds, accepts; if does not find, rejects. If $l \geq \mu$, then U_L searches the periodic part. First, in order to adjust to the modular behavior in the periodic part, U_L decreases l by the length μ. Then, it tests whether this decreased l is directly in the period λ by by checking if $l < \lambda$. If $l \geq \lambda$, U_L take the modulo λ of l. Finally, it searches the periodic part for the resulting number $l' \equiv (l - \mu) \pmod{\lambda}$: If it finds, accepts; if does not find, rejects. Thus, UFPFs can decide URLs. One may want to note that using the modulo λ operation (done by using long division) in the periodic part and binary search in both searches (i.e. in non-periodic part or periodic part) should result in $O(\log l + \log l) = O(\log l)$ running time (assuming that comparing/adding/subtracting two g-bit numbers for some fixed g takes $O(1)$ runnning time and considering that the input length is $\lceil \log l \rceil$).

Now, we will show that every UFPF has an equivalent UDFA by constructing a UDFA from a given UFPF and this construction can be done easily.

Suppose a UFPF $U_L = (a, \mu, \lambda, [\mu_1, \mu_2, ..., \mu_m], [\lambda_1, \lambda_2, ..., \lambda_n]\}$ is given for some URL L. Then, we can give an equivalent UDFA $D_L = \{\{q_0, q_1, ..., q_{\mu+\lambda-1}\}, \{a\}, \delta, q_0, F\}$ where

- $\delta(q_i, a) = q_{i+1}$ for $0 \leq i \leq \mu + \lambda - 2$,
- $\delta(q_{\mu+\lambda-1}, a) = q_\mu$,
- $F = \{q_i \mid i = \mu_x \text{ for } 1 \leq x \leq m \text{ or } i = \mu + \lambda_y \text{ for } 1 \leq y \leq n\}$.

Clearly, D_L (only) accepts strings a^{μ_x} (by the tail or transient part of D_L) and $a^{\mu+k\lambda+\lambda_y}$ (by the cycle or periodic part of D_L) where $k \in \mathbb{N}$, and decides L.

Related to the proof above, some observations are presented below:

- Generating UDFA from UFPFs is an easily revertible process. Since we assume that UDFA are increasingly sorted, it is adequate to highlight the accepted states, their enumerations, and the periodic behaviour of a given UDFA to generate its associated UFPF.
- Generating UDFA from UFPFs can be costly because the number of periodic or transient members in a UFPF can be logarithmically smaller than its period and transience. Since we keep data about every state and their transitions in UDFA, it can be exponentially costly to construct UDFA from UFPFs. For example, consider the URL $A_m = \{a^m\}$ which has only one member that has length m. Its associated UFPF is $U_{A_m} = (a, m+1, 1, [m], [])$, and its deciding UDFA has $m+2$ states. It is logarithmically less costly to represent A_m with a UFPF rather than with a UDFA.

5.3. Introducing Compact Unary Finite Periodic Form

Now, we will analyze the storage needs of UFPFs. Suppose a unary finite periodic form $U_L = (a, \mu, \lambda, [\mu_1, \mu_2, ..., \mu_m], [\lambda_1, \lambda_2, ..., \lambda_n])$ is given, having m and n members in its non-periodic part and periodic part, respectively. Then, the stored data inside this UFPF takes nearly $m\lceil \log \mu \rceil + n\lceil \log \lambda \rceil$ bits. If m and n is very small compared to μ

and λ, respectively, then we can say that the storage is space-efficient. An example is the UFPF $U_{A_m} = (a, m+1, 1, [m], [])$ for the URL $A_m = \{a^m\}$, where U_{A_m} takes only logarithmic space (logarithmic in m).

On the other hand, if $m \approx \mu$ or $n \approx \lambda$, the storage space is very close to $\mu\lceil\log\mu\rceil + \lambda\lceil\log\lambda\rceil$. To be more efficient in data storage, we can improve our UFPF model. We claim that we can hold the storage need up to $F = \mu + \lambda$ by changing our form accordingly to the conditions stated below. First, we will define this more compact form:

Definition 5.6. The *compact unary finite periodic form*, shortly *CUFPF*, is an alternate UFPF where every non-periodic member or periodic member is represented by only 1 bit, instead of logarithmic number of bits. Precisely, for a given UFPF $U_L = (a, \mu, \lambda, [\mu_1, ..., \mu_m], [\lambda_1, ..., \lambda_n])$ deciding the URL L, the equivalent CUFPF is $Z_L = (a, \mu, \lambda, M, N)$ where

- $M = x_0x_1...x_{\mu-1}$ is a μ-bit binary word-number called the *transience word* where x_i are bits: $x_i = 1$ if and only if $a^i \in L$ and $x_i = 0$ if and only if $a^i \notin L$,
- $N = y_0y_1...y_{\lambda-1}$ is a λ-bit binary word-number called the *period word* where y_i are bits: $y_j = 1$ if and only if $a^{\mu+k\lambda+j} \in L$ and $y_j = 0$ if and only if $a^{\mu+k\lambda+j} \notin L$ for $k \in \mathbb{N}$.

Basically, by using CUFPFs, we are representing a member of a URL with only 1 bit; so one should note that the size of M is $|M| = \mu$ and the size of N is $|N| = \lambda$. We can also use CUFPFs to decide URLs:

Suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for the URL L where $M = x_0x_1...x_{\mu-1}$ and $N = y_0y_1...y_{\lambda-1}$. To check whether a string $a^p \in L$, we look at the bit x_p of M is 1 or 0 if $p < \mu$. If it is 1, we accept; otherwise, we reject. If $p \geq \mu$, first we make p sufficiently small, by substracting first μ and doing the modulo λ operation on it to make p a non-negative integer p' which is smaller than λ. Then, we check if the bit $y_{p'}$ of N is 1 or 0. If it is 1, we accept; if not, we reject. Thus, CUFPFs can decide

URLs.

Now, we will give the conditions that using CUFPFs instead of UFPFs to describe URLs is more efficient in data storage:

Theorem 5.1. Given UFPF $U_L = (a, \mu, \lambda, [\mu_1, ..., \mu_m], [\lambda_1, ..., \lambda_n])$ and its associated CUFPF, named as $Z_L = (a, \mu, \lambda, M, N)$ deciding the same URL L, to limit the total storage space usage for transient members and periodic members by $\mu + \lambda$ bits:

- If $m > \frac{\mu}{\lceil \log \mu \rceil}$ and $n > \frac{\lambda}{\lceil \log \lambda \rceil}$, use CUFPF Z_L.
- If $m < \frac{\mu}{\lceil \log \mu \rceil}$ and $n < \frac{\lambda}{\lceil \log \lambda \rceil}$, use UFPF U_L.
- Else, use the more efficient one which naturally is limited above by $\mu + \lambda$ bits.

Proof. The proof is simple. We want to limit storage space by $F = \mu + \lambda$. Thus, if $m > \frac{\mu}{\lceil \log \mu \rceil}$, the storage space for transient members of U_L takes $P = m\lceil \log \mu \rceil > \frac{\mu}{\lceil \log \mu \rceil}\lceil \log \mu \rceil = \mu$ bits and that of Z_L takes μ bits. For the periodic members, if $n > \frac{\lambda}{\lceil \log \lambda \rceil}$, the storage space takes $R = n\lceil \log \lambda \rceil > \frac{\lambda}{\lceil \log \lambda \rceil}\lceil \log \lambda \rceil = \lambda$ bits for U_L and λ bits for Z_L. Thus, using Z_L for the case where $m > \frac{\mu}{\lceil \log \mu \rceil}$ and $n > \frac{\lambda}{\lceil \log \lambda \rceil}$ limits the storage use by $\mu + \lambda$ bits, instead of $P + R > \mu + \lambda$ when using U_L. On the other hand, if $m < \frac{\mu}{\lceil \log \mu \rceil}$ and $n < \frac{\lambda}{\lceil \log \lambda \rceil}$, then $P + R < \frac{\mu}{\lceil \log \mu \rceil}\lceil \log \mu \rceil + \frac{\lambda}{\lceil \log \lambda \rceil}\lceil \log \lambda \rceil = \mu + \lambda$; hence using U_L instead of Z_L becomes the better option. One can see that last case is trivial. □

From now on, we will work with CUFPFs instead of UFPFs unless we say otherwise. Next, we will show how to minimize CUFPFs.

5.4. Minimization of CUFPFs

In this section, we will focus on how to minimize CUFPFs. First, we will begin with the definition of minimal CUFPFs:

Definition 5.7. A CUFPF Z_L, which describes the URL L, with transience μ and period λ is said to be *minimal* if there does not exist another CUFPF, which describes the same L, with transience ν and period γ such that $\nu + \gamma < \mu + \lambda$.

One should note that this definition of minimality implies that there can be more than one minimal CUFPF deciding the same language. Now, we will prove that given any URL L, there is only one minimal CUFPF deciding L, but before that we will give a definition and two theorems that will be useful for this proof:

Definition 5.8. A non-empty word w is said to be *primitive* if it is not the case that $w = s^k$ for some word s and an integer $k \geq 2$ [29].

Theorem 5.2. For every non-empty word w, $w = s^k$ for some unique primitive word s and unique integer $k \geq 1$ [29].

Theorem 5.3. Given two words s_1 and s_2, if $s_1 s_2$ is primitive, then $s_2 s_1$ is also primitive [30].

Now, we will prove the following:

Theorem 5.4. Given any URL L, there is only one minimal CUFPF deciding L.

Proof. Suppose there exists two CUFPFs deciding the same URL $L \in \{a\}^*$, namely $Z_1 = (a, \mu, \lambda, M_1, N_1)$ and $Z_2 = (a, \nu, \gamma, M_2, N_2)$ where $\mu + \lambda = \nu + \gamma = F$. Suppose $\mu > \nu$ and $\lambda < \gamma$. We now prove that such Z_1 and Z_2 cannot coexist.

Say N_2 consists of γ bits such that $N_2 = b_1 b_2 ... b_\gamma$. One can see that there is another CUFPF $Z_3 = (a, \mu, \gamma, M_1, N_3)$ that decides the same language with N_3 being a circularly k-bit left-shifted version of N_2:

$$N_3 = b_k b_{k+1} ... b_\gamma b_1 b_2 ... b_{k-1} \text{ where } k \equiv \mu - \nu \pmod{\gamma} \text{ and } k < \gamma.$$

For Z_3 to decide L, it should be that $N_3 = N_1 t$ for some non-empty word t and $N_1^\gamma = N_3^\lambda$. Furthermore, we have the following:

- N_1 cannot be of the form $N_1 = u_1^{k_1}$ for some word u_1 and integer $k_1 \geq 2$. Otherwise, $F = \mu + \lambda$ would not be minimal.
- N_2 cannot be of the form $N_2 = u_2^{k_2}$ for some word u_2 and integer $k_2 \geq 2$. Otherwise, $F = \nu + \gamma$ would not be minimal.
- N_3 cannot be of the form $N_3 = u_3^{k_3}$ for some word u_3 and some integer $k_3 \geq 2$. Otherwise, there would be another word v such that $N_2 = v^{k_3}$ and $F = \nu + \gamma$ would not be minimal.

From definition 5.8, we know that N_1, N_2, and N_3 are all binary primitive words. Using theorem 5.2 with our knowledge about N_1, N_2, and N_3 gives us that

$$N_1^{\gamma} \neq N_3^{\lambda}$$

and this contradicts what we have said earlier. Hence, Z_1 and Z_3, and thus Z_1 and Z_2 cannot both decide L. Therefore, there can exist only 1 minimal CUFPF for a URL. □

A direct result of this theorem 5.4 is the following:

Theorem 5.5. A CUFPF Z_L that decides L, having transience word M and period word N, is a minimal CUFPF if and only if it has the following properties:

- N is a primitive word.
- It is not the case that $M = PS$ and $N = RS$ for some non-empty word S and words P and R (i.e. M and N do not share any non-empty suffix).

Proof. For the forward direction of the theorem, suppose Z_L is minimal. We will show that it should have those two properties:

- Suppose N is not a primitive word. Then, there exists another word W such that $N = W^k$ for some $k \geq 2$. Then, there exists another CUFPF deciding L with

transience word M and period word W. One can see that $|M|+|W| < |M|+|N|$, so Z_L cannot be minimal. Thus, N must be a primitive word.

- Suppose there exists a non-empty word S such that $M = PS$ and $N = RS$ where P and R are words. Then, there exists another CUFPF deciding L with transience word P and period word SR. One can see that $|P|+|SR| < |M|+|N|$, so Z_L cannot be minimal. Thus, it is not the case that $M = PS$ and $N = RS$ for some non-empty word S and words P and R.

Before proving the other direction of this theorem, we want to highlight a feature of CUFPFs. A CUFPF can also be seen as a tool that generates an infinite sequence of binary numbers, which at some point shows a periodic behaviour, and this is directly related to the "ultimately periodic" characteristic of URLs. Sequence generation procedure of a CUFPF is simple: First it gives its transience word, then it concatenates infinitely many copies of its period word to that. One can see that to decide the same URL, two different CUFPFs should be able generate the same infinite sequence; otherwise there would exist some non-negative integer i such that some element of L with length i is accepted in one of the CUFPFs and rejected in the other one, giving a contradiction. Now, we continue our proof.

For the other direction of the theorem, we will show that if M and N do not share any non-empty suffix and N is primitive, then Z_L is minimal. Now, suppose that some minimal CUFPF Z'_L that decides L has transience word U and period word V. Since both Z_L and Z'_L decide the same language, they should generate the same infinite sequence. We will use this "same sequence" logic to handle two scenarios to conclude that $M = U$ and $N = V$:

For the first scenario, say $|M| > |U|$. Then, we can state the following: $M = UV^{k_2}x_2$ for some integer k_2 and a prefix x_2 of V (i.e. the sequences should be the same upto $|M|^{th}$ bit), and $V = x_2y_2$ for words x_2 and y_2. Moreover, we can say that $MN^{|V'|} = UV^{k_2}x_2V'^{|N|}$ (i.e. upto $|M| + |V'||N|^{th}$ symbol, the sequences should be the same) where $V' = y_2x_2$, implying $V'^{|N|} = N^{|V'|} = N^{|V|}$. By theorem 5.3, since V is primitive, V' is also primitive. Using theorem 5.2 gives us that $N = V' = y_2x_2$ since

both N and V' are primitive words. Then, M and N share the same suffix x_2, giving a contradiction. Thus, $|M| \not> |U|$.

Say $|M| \leq |U|$ for the second scenario. Since there does not exist any CUFPF that can decide L with a transience word M such that $|M| < |U|$, $|M| \not< |U|$. That means $|M| = |U|$, implying $M = U$. By a similar argument about primitive words used in the previous scenario (i.e. $N^{|V|} = V^{|N|}$), it should be that $N = V$. Henceforth, Z_L is equal to Z'_L, and Z_L is minimal. The proof is done. □

Now, to minimize a given CUFPF $Z_L = (a, \mu, \lambda, M, N)$, we will minimize its period and minimize its transience, and make sure that this minimized version has the properties that are stated in theorem 5.5. Now, we will prove the following:

Theorem 5.6. Suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for some URL L. It takes $O(\lambda^2 + \mu)$ steps to minimize Z_L.

Proof. To correctly minimize Z_L, we should accomplish the following three tasks:

(i) We should check whether the period word N is of the form $N = VV...V = V^k$ for some primitive word V and some positive integer k. V becomes our new period word.

(ii) We should check if the transience word M is of the form $M = UVV...V = UV^l$ for some integer l and some word U. If it is the case, we should find the shortest such U such that V is not a suffix of U. U becomes our new transience word.

(iii) We should check whether $U = PS$ and $V = RS$ for some non-empty word S and words P and R. If it is the case, we should find the longest such S, and hence, shortest such P. P becomes our minimal transience word and $W = SR$ becomes our minimal period word.

These three tasks all together guarantee that the minimized CUFPF has all the properties stated in theorem 5.5 and is actually minimal. Now, we will show how to

minimize a CUFPF by giving the exact reduction procedures on both of its words. Furthermore, we will show that these operations can be done efficiently:

First, we do the first task. Basically, we check if $N = VV...V = V^k$ for some primitive word V which is also the first $|V|$ bits of N. We do this starting from the shortest possible V (i.e. $|V| = 1$, the first bit of N), and continue until we find such a match, at most upto $|V| = \lceil |N|/2 \rceil$. If such V is found, we stop and say V is the new period word; otherwise we say $N = V$ and is still the period word. At every step, we do at most $(k-1)|V| < |N|$ bitwise comparisons, and $\lceil |N|/2 \rceil$ steps exist at maximum; hence, this part takes $O(|N|^2) = O(\lambda^2)$ steps.

Now, we do the second task. Basically, we check if $M = UVV...V = UV^l$ for some word U that is the first $|U|$ bits of M. At the beginning, we check if V is a suffix of M and if that is not the case, we stop and say $M = U$ and is still the transience word. Otherwise, we erase the last $|V|$ bits of M. At every step, we continue to erase the last $|V|$ bits of the remaining prefix of M if V is still a suffix of it. On some prefix U of M, if such a deletion operation cannot occur, we stop and say U is the new transience word. At every step, we do $|V|$ bitwise comparisons (and bit deletions, if possible), and $\lceil |M|/|V| \rceil$ steps exist at maximum; hence, this part takes $O(|M|) = O(\mu)$ steps.

Finally, we do the third task. Basically, we want to find the longest word S such that $U = PS$ and $V = RS$. To do that, starting from their respective last bits, we pairwise compare the i^{th} bits of U and V at every step i (i^{th} bits from their ends), until a mismatch occurs. Such a mismatch tells us where to stop and we find the longest shared suffix of U and V, say S. Then, we stop and say P is the transience word and W is the period word of the minimal CUFPF that decides L where $U = PS$ and $W = SR$ for that word S. At every step, we do one bitwise comparison, and $|V|$ steps exist at maximum; hence, this part takes $O(|V|) = O(\lambda)$ steps.

The resulting CUFPF $Z'_L = (a, |P|, |W|, P, W)$ is minimal, because it has the properties of minimal CUFPFs. Furthermore, to obtain this minimal CUFPF, our algorithm first finds the shortest period possible and shortest period word possible (in

perhaps a circularly shifted order), then it finds the shortest transience word possible and the shortest period word (by checking if it is circularly shifted or not). Therefore, we can minimize the CUFPF $Z_L = (a, \mu, \lambda, M, N)$ in $O(\lambda^2 + \mu)$ steps (of bitwise comparison, deletion, and writing operations). □

Minimization of DFA in general is a relatively old concept. Several efficient algorithms accomplishing this task are known and one is Hopcroft's DFA minimization algorithm given in Hopcroft's paper [31]. In the worst case, this algorithm runs in $O(ns \log n)$ steps (of basic state comparison/addition/deletion), where n is the number of states and s is the size of the alphabet. Thus, for unary case, it runs in $O(n \log n)$ steps. A quick comparison shows that the algorithm given for CUFPF minimization in the proof of theorem 5.6 is not better than Hopcroft's technique in the number of steps of basic operations, in the worst case (i.e when $\lambda \gg \mu$, $\lambda \to n$ and $O(n^2) \geq O(n \log n)$ where $n = \lambda + \mu$). However, there are also cases it can perform as well as Hopcroft's technique (i.e. when $\mu \gg \lambda$, $\mu \to n$ and $O(n) \leq O(n \log n)$ where $n = \lambda + \mu$).

By the way, the procedure described in the first task can be also called as period shrinking because it shrinks periodic parts of CUFPFs. Furthermore, the procedures described in the second and third tasks together can be also called as transience shrinking because it shrinks transient parts of CUFPFs. These notions will be useful when we deal with union and intersection operations on CUFPFs. Finishing minimization, now, we will move onto the closure properties of CUFPFs.

5.5. Closure Properties

CUFPFs are closed under complementation, union, intersection, concatenation, and star operations since the class of regular languages are closed another these operations. Usually, arguments involving DFA and NFA are used to prove these closure properties. In this section, we will use CUFPFs for that matter and show how many bits (bits as symbols of words used in CUFPFs) and/or steps of bitwise operations are needed for such operations.

5.5.1. Complementation

Complementation is rather a simple operation. The complement of any regular language L, namely $\bar{L}$, is also regular. Say a DFA D_L decides L. To show that $\bar{L}$ is also regular, it is enough to construct a DFA $D_{\bar{L}}$ that decides $\bar{L}$. This can be done by making the final states of D_L the non-final states of $D_{\bar{L}}$ and the non-final states of D_L the final states of $D_{\bar{L}}$, while keeping the states, alphabet, transition function, and the starting state of D_L the same in $D_{\bar{L}}$. Now, we will use this approach in complementation of CUFPFs.

Theorem 5.7. CUFPFs are closed under the complement operation. Furthermore, suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for a language L, it takes

$$O(\mu + \lambda)$$

steps of basic bitwise operations to construct the complement $Z_{\bar{L}}$ of Z_L. $Z_{\bar{L}}$ has a transience of μ and a period of λ.

Proof. Suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for a URL L. We now prove that a CUFPF $Z_{\bar{L}}$, which describes $\bar{L}$, can be constructed by using Z_L. This is simple: One by one, we only flip the bits x_is and y_js of $M = x_0x_1...x_{\mu-1}$ and $N = y_0y_1...y_{\lambda-1}$. Indeed, since $x_i = 1$ if and only if $a^i \in L$ and $y_j = 1$ if and only if $a^{\mu+j} \in L$, the flipped bits $x'_i = 1$ if and only if $a^i \notin L$ (or $a^i \in \bar{L}$) and $y'_j = 1$ if and only if $a^{\mu+j} \notin L$ (or $a^{\mu+j} \in \bar{L}$). This procedure clearly takes $O(\mu + \lambda)$ steps (of bitwise flips). □

5.5.2. Union and Intersection

The class of regular languages is closed under union and intersection operations. This can be easily proven and here is a sketch of proof for that:

We want to show that if L_1 and L_2 are regular languages, so is $L_1 \cup L_2$ (respectively, $L_1 \cap L_2$). Therefore, we should show that there exists a DFA that decides $L_1 \cup L_2$ (resp.

$L_1 \cap L_2$). Since there exist two DFA D_1 and D_2 that decides L_1 and L_2, respectively, one can construct another DFA D_3 that simulates D_1 and D_2 at the same time while reading an input. To simulate D_1 and D_2, at any step, D_3 should remember a pair of states that D_1 and D_2 are on at that step, so the states of D_3 should actually be the Cartesian product of the states of D_1 and D_2. By also obeying the rules about the transition functions and final states of both D_1 and D_2 and union (resp. intersection) operation, one can construct such an intended D_3.

In general case, union (or resp. intersection) of two DFA with m and n states result in an mn-state DFA. In unary case, union and intersection are analyzed with great detail in Shallit's paper [32], and the main results of this work is given in the following:

Theorem 5.8. Let D_1 and D_2 be two UDFA, deciding L_1 and L_2 respectively. Suppose the transition diagram of D_1 (resp. D_2) has a tail of size t and a cycle of size c (resp. t', c'). Then the state complexity of $L_1 \cap L_2$ (or resp. $L_1 \cup L_2$) is not greater than $\max(t, t') + \text{lcm}(c, c')$ [32] .

Now, we analyze union and intersection operations with CUFPFs. First, we introduce two lemmas that will be helpful for the union and intersection operations.

Lemma 5.9. Suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for the language L. For some integer $f > \mu$, there exists a *μ_f-stretched* Z_L, shortly μ_f-Z_L, deciding the same L such that μ_f-$Z_L = (a, f, \lambda, M', N')$ where

- $M' = MN^k v$ and $f = |M'| = |M| + k|N| + |v|$ for some non-negative integer k and a prefix v of N,
- $N' = uv$ where $N = vu$ for some word u.

This is called transience stretching.

Proof. Basically, we want to generate a CUFPF that decides the same URL Z_L and has transience f and period λ. Thus, we add $f - \mu$ bits to the right of M and create

a new transience word M' and change the order of bits of N by some circular modulo operation and create a (possibly) new period word N' which has the same length as N. Strictly speaking:

- $M' = MN^k v$ where $k|N| + |v| = |M'| - |M| = f - \mu$ for some non-negative integer k and a prefix v of N,
- $N' = uv$ where $N = vu$ for some word u.

The definition of M' and N' should be easily understandable, otherwise, one may want to recall the notion of CUFPF minimization and the fact that equivalent CUFPFs generate "same sequences" (see theorem 5.5 in CUFPF minimization part). Here, instead of reducing or shrinking a word, we stretch it. □

Lemma 5.10. Suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for some language L. For some positive integer $p = r\lambda$ where $r > 1$, there exists a λ_p-*stretched* Z_L, shortly λ_p-Z_L, deciding the same L such that λ_p-$Z_L = (a, \mu, p, M, N')$ where $N' = NN...N = N^p$. This is called period stretching.

Proof. Basically, we want to generate a CUFPF that decides the same URL Z_L and has transience μ and period λ^p. Thus, talking in the language of CUFPFs, we add $(p-1)$ Ns to right of N and create a new period word $N' = N^p$. □

Knowing these two lemmas, now suppose a CUFPF $Z_L = (a, \mu, \lambda, M, N)$ is given for a language L. Given two positive integers $f > \mu$ and $p = r\lambda$ for some $r > 1$, we can doubly stretch Z_L and construct another CUFPF called as μ_f-λ_p-*stretched* Z_L, shortly μ_f-λ_p-Z_L, such that μ_f-λ_p-Z_L decides the same L. To do that, one should first do transience stretching on Z_L by f and then do period stretching on μ_f-Z_L by p. We will use this doubly-stretched CUFPF notion in the following theorem.

Theorem 5.11. CUFPFs are closed under the union (respectively, intersection) operation. Moreover, suppose CUFPFs $Z_{L_1} = (a, \mu, \lambda, M_1, N_1)$ and $Z_{L_2} = (a, \nu, \gamma, M_2, N_2)$

are given for two languages L_1 and L_2 respectively, it takes

$$O(\max(\mu, \nu) + \text{lcm}(\lambda, \gamma))$$

steps of basic bitwise operations to construct the union (resp. intersection) Z_{L_3} of Z_{L_1} and Z_{L_2}. Z_{L_3} has a transience of $\xi = \max(\mu, \nu)$ and a period of $\rho = \text{lcm}(\lambda, \gamma)$.

Proof. The CUFPFs $Z_{L_1} = (a, \mu, \lambda, M_1, N_1)$ and $Z_{L_2} = (a, \nu, \gamma, M_2, N_2)$ are given for languages L_1 and L_2. We claim that there exists a CUFPF $Z_{L_3} = (a, \xi, \rho, M_3, N_3)$ where $\xi = \max(\mu, \nu)$ and $\rho = \text{lcm}(\lambda, \gamma)$ that describes L_3 where $L_3 = L_1 \cup L_2$ (M_3, N_3 are going to be described shortly after). We can construct Z_{L_3} from Z_{L_1} and Z_{L_2}. First, we stretch Z_{L_1} and Z_{L_2} accordingly because instead of them, we will use their stretched versions. If $\nu > \mu$, we formally use the CUFPFs μ_ν-λ_ρ-$Z_{L_1} = (a, \nu, \rho, M_1', N_1')$ and γ_ρ-$Z_{L_2} = (a, \nu, \rho, M_2, N_2')$, otherwise we use λ_ρ-$Z_{L_1} = (a, \mu, \rho, M_1, N_2')$ and ν_μ-γ_ρ-$Z_{L_2} = (a, \mu, \rho, M_2', N_2')$. Such stretching operations will take a total of $O(\xi + \rho)$ steps.

For the transient part: Suppose that $\nu > \mu$. We now merge the transient parts of L_1 and L_2 in CUFPF manners. For that, we take every i^{th} bit of M_1' and M_2 and do the following for union operation (resp. intersection operation): If either one is 1 (resp. 0), set the i^{th} bit of M_3 to 1 (resp. 0); else, set the i^{th} bit of M_3 to 0 (resp. 1). A very similar argument holds with λ_ρ-Z_{L_1} and ν_μ-γ_ρ-Z_{L_2} when $\nu \leq \mu$. That way, we guarantee to describe L_3's every member whose length is shorter than ξ, obeying the transience rules of L_1 or (resp. and) L_2. This should take $O(\xi)$ steps.

For the periodic part: The new period is $\rho = \text{lcm}(\lambda, \gamma)$. We now merge the periodic parts of L_1 and L_2 in CUFPF manners. For that, we take every i^{th} bit of N_1' and N_2' and do the following for union operation (resp. intersection operation): If either one is 1 (resp. 0), set the i^{th} bit of N_3 to 1 (resp. 0); else, set the i^{th} bit of N_3 to 0 (resp. 1). That way, we guarantee to describe L_3's every member whose length is longer than or equal to ξ, obeying the periodicity rules of L_1 or (resp. and) L_2. This

should take $O(\rho)$ steps.

By stretching Z_{L_1} and Z_{L_2} and merging them accordingly as stated above, we can create Z_{L_3} deciding L_3, the union (resp. intersection) of L_1 and L_2. Hence, CUFPFs are closed under the union (resp. intersection) operation and this procedure costs $O(\max(\mu, \nu) + \text{lcm}(\lambda, \gamma))$ steps (of bitwise comparison and writing operations). □

As one might easily notice, these results about union and intersection operations with CUFPFs are compatible with the ones about union and intersection of URLs given in theorem 5.8.

Now, we will shortly show that the state bound in both cases, $O(\max(\mu, \nu) + \text{lcm}(\lambda, \gamma))$, and especially the less trivial part $O(\text{lcm}(\lambda, \gamma))$, cannot be improved in the general case.

Given languages $B_\lambda = \{a^{k\lambda} \mid k \in \mathbb{N}\}$ and $B_\gamma = \{a^{k\gamma} \mid k \in \mathbb{N}\}$, where λ and γ are both prime numbers, we know the following (by theorem 4.6):

- the minimal DFA deciding B_λ needs λ states,
- the minimal DFA deciding B_γ needs γ states,
- the minimal DFA deciding $B_\lambda \cap B_\gamma$ needs $\lambda\gamma$ states, where $B_\lambda \cap B_\gamma = \{a^{k\lambda\gamma} \mid k \in \mathbb{N}\}$.

Thus, the minimal CUFPFs deciding languages B_λ, B_γ, and $B_\lambda \cap B_\gamma$ have periods λ, γ, and $\lambda\gamma = \text{lcm}(\lambda, \gamma)$, respectively. Hence, the bound $O(\max(\mu, \nu) + \text{lcm}(\lambda, \gamma))$ cannot be improved in intersection case. One can also check the bound for union case, by considering the complements of B_λ and B_γ, and the equality that $\bar{B}_\lambda \cup \bar{B}_\gamma = \overline{B_\lambda \cap B_\gamma}$.

5.5.3. Star

As stated in Yu, Zhuang, and Salomaa's paper "The state complexities of some basic operations on regular languages" (1994) [33], given an n-state UDFA deciding some URL L, there exists a UDFA that decides L^* with no more than $(n-1)^2+1$ states. First we, will analyze this result and show how to explicitly find such a UDFA deciding L^*. Then, we will use this knowledge to show the closure of star operation on CUFPFs.

Theorem 5.12. Let D_L be an n-state UDFA that decides L. Then, there exists a UDFA D_{L^*} that decides L^* with no more than $(n-1)^2+1$ states [33]. Furthermore, fully building D_{L^*} takes $O(n^4)$ steps of basic elementwise operations (i.e. basic comparison, addition, deletion operations on states).

Proof. Let D_L be the n-state UDFA such that $D_L = (Q, \{a\}, \delta, q_0, F)$ where $n = \mu+\lambda$, $Q = \{q_0, q_1, ..., q_{\mu-1}, q_\mu, q_{\mu+1}, ..., q_{\mu+\lambda-1}\}$, $\delta(q_{\mu+\lambda-1}, a) = q_\mu$, and $\delta(q_i, a) = q_{i+1}$ for every $i \leq n-2$. If q_0 is the only final state of D_L, then $L^* = L$, and the proof is trivial. If not, then there is at least one final state that is different from q_0. Using D_L, we can construct an NFA N_{L^*} such that $N_{L^*} = (Q, \{a\}_\epsilon, \delta', q_0, F')$ where $\delta' = \delta \cup \{(q, \epsilon, q_0) \mid q \in F\}$ and $F' = F \cup q_0$. We now explain why N_{L^*} decides L^*.

For an NFA to decide L^*, it should accept every word w such that $w = w_1 w_2 ... w_k$ where $k \geq 0$ and $w_i \in L$ for every $i \leq k$. By securing a return to its start state after reading every such w_i, an NFA can accept w. Thus, for every final state of D_L, if we add an ϵ arrow from that state to the start state of D_L, we guarantee such a return. Our intended NFA should also accept the empty word, thus we should make q_0 a final state (if it was not before in D_L). As a result of these additions, our NFA N_L decides L^*. Now, by modifying N_{L^*} and using subset construction approach (details are given in NFA-to-DFA conversion theorem 3.1), we will construct a UDFA D_{L^*} which simulates N_{L^*} and decides L^*.

Our UDFA decider for L^*, D_{L^*} is a 5-tuple such that $D_{L^*} = (P, \{a\}, \eta, \{q_0\}, F_P)$ and it has the following properties:

(i) $P \subseteq \mathcal{P}(Q)$
(ii) For any $X \in P$, $\eta(X, a) = \{q \in Q \mid \exists p \in X,\ \delta'(p, a) = q \vee \delta'(\delta'(p, \epsilon), a) = q\}$.
(iii) $F_P = \{X \in P \mid X \cap F \neq \emptyset \vee X = \{q_0\}\}$

All those properties are a direct result of using the conversion procedure described in theorem 3.1. Now, we show that a UDFA having these three properties can actually decide L^* with no more than $(n-1)^2 + 1$ states.

Let q_f be the first final state that can be reachable from q_0 in D_L and let a^z be the shortest word such that "the computation of D_L on word a^z hits q_f", denoted by $COMP(D_L, q_0, a^z) = q_f$ (see that $z \leq n-1$). We know that a computation of N_{L^*} on word a^z hits some $q \in \{q_0, q_f\}$, so the computation of D_{L^*} on word a^z should hit $\{q_0, q_f\}$ since D_{L^*} simulates N_{L^*}. Thus, $COMP(D_{L^*}, \{q_0\}, a^z) = \{q_0, q_f\}$. Moreover, let us denote the state $COMP(D_{L^*}, \{q_0\}, a^{iz}) \in P$ by p_{k_i} for $i \geq 0$.

In the paper [33], it is claimed that $p_{k_i} \supseteq p_{k_{i-1}}$ for all $i \geq 1$ since $p_{k_1} = COMP(D_{L^*}, \{q_0\}, a^z) = \{q_0, q_f\}$ and:

$$\begin{aligned} p_{k_i} &= COMP(D_{L^*}, \{q_0\}, a^{iz}) \\ &= COMP(D_{L^*}, \{q_0, q_f\}, a^{(i-1)z}) \\ &= COMP(D_{L^*}, \{q_0\}, a^{(i-1)z}) \cup COMP(D_{L^*}, \{q_f\}, a^{(i-1)z}) \\ &= p_{k_{i-1}} \cup COMP(D_{L^*}, \{q_f\}, a^{(i-1)z}) \end{aligned} \tag{5.1}$$

There are two possibilities: Either $p_{k_i} = p_{k_{i-1}} \subseteq Q$ for some $i \leq n-1$, or $p_{k_{n-1}} = Q$. The first case is trivial. For the other case, if $p_{k_i} \neq p_{k_{i-1}}$ for any such i, then $p_{k_{n-1}}$ should contain at least n states of Q. Since $p_{k_{n-1}}$ cannnot contain more states than n, it must be equal to Q, implying that $\eta(p_{k_{n-1}}, a) = p_{k_{n-1}}$. In either case, D_{L^*} cannot have more than $z(n-1)+1$ states. Now, we see why is that:

- Say $p_{k_i} = p_{k_{i-1}} \subseteq Q$ for some $i \leq n-1$. Say a^{iz} is given as input. Then, on its prefix $a^{(i-1)z}$, D_{L^*} hits at most $(i-1)z+1$ different states, last one being $p_{k_{i-1}}$. Then, on the remaining suffix a^z, D_{L^*} hits at most $z-1$ different states since the last state is $p_{k_i} = p_{k_{i-1}}$. This implies that periodic part of D_{L^*} has a period of length z and D_L has at most $(i-1)z+1+z-1 = iz \leq (n-1)z$ states.
- Say $p_{k_{n-1}} = Q$. We know that $\eta(p_{k_{n-1}}, a) = p_{k_{n-1}}$. Thus, on input $a^{(n-1)z+k}$ where $k \geq 1$, D_{L^*} hits at most $(n-1)z+1$ different states on prefix $a^{(n-1)z}$ and then it goes into a loop of length 1 for the remaining k symbols. Thus, D_L has at most $(n-1)z+1$ states.

We know that $z \leq n-1$; thus, there exists a UDFA that decides L^* with no more than $(n-1)^2+1$ states.

Now, we turn our attention to explicitly creating the states and transition function of D_{L^*}. We know that to simulate N_{L^*} on some input is actually to remember the states it hits at every step. Since D_{L^*} can decide L^* with no more than $(n-1)^2+1$ states, it is enough to remember the states that N_{L^*} can hit on input $a^{(n-1)^2}$. Clearly, this whole state creation procedure for D_{L^*} can be done in $(n-1)^2$ steps by remembering at most n states and increasingly enumerating them at every step. With this procedure, one can also explicitly give the transition rules for the states D_{L^*}, except the one for the final state enumerated. To find the state where the periodic part of D_{L^*} starts, it should be enough to find the last step j such that the enumerated state in step j is equal to the final state (i.e. N_{L^*} hits the same set of states at step j and step $(n-1)^2$).

Now, we should give a method to pinpoint the final state set F_P of D_{L^*}. We know that $|P| \leq (n-1)^2+1$ and $F_P = \{X \in P \mid X \cap F \neq \emptyset \vee X = \{q_0\}\}$. Thus, for every $X \in P$, we should check if $X \cap F \neq \emptyset$ or not. Since $|F| \leq n$ and every $|X| \leq n$, every such check can be performed in $O(n^2)$ steps of pairwise comparisons (i.e. for some $x \in X$ and $f \in F$, we check if $x = f$). There are $(n-1)^2+1$ different comparisons at maximum; so we can find the final states of D_{L^*} in a total of $O(n^4)$ steps. Therefore, fully building D_{L^*}, which decides L^* takes $O(n^4)$ steps of basic elementwise operations. □

It is known that for any $n \geq 1$, there exists an n-state minimal UDFA that decides some URL L and an $(n-1)^2+1$-state minimal UDFA that decides L^* [33]; hence, the state bound is optimal. Now, we move onto CUFPFs.

Theorem 5.13. CUFPFs are closed under the star operation. Furthermore, suppose a CUFPF $Z_L = \{a, \mu, \lambda, M, N\}$ is given for some language L, it takes

$$O((\mu+\lambda)^4)$$

steps of elementwise operations to construct Z_{L^*}. The sum of the transience and period of Z_{L^*} is $(\mu+\lambda-1)^2+1$.

Proof. The proof is simple. First, we first construct Z_L's equivalent UDFA D_L. Then, using theorem 5.12, we construct an $(n-1)^2+1$-state UDFA D_{L^*} that decides L^*. Finally, we generate D_{L^*}'s equivalent CUFPF Z_{L^*} for L^*. By theorem 5.12, we know that creating the states (with transition rules) and final states of D_{L^*} takes $O(n^2)$ and $O(n^4)$ steps, respectively. We also know that converting Z_L to D_L and D_{L^*} to Z_{L^*} should take $O(n)$ and $O(n^2)$ steps, respectively. Hence, it should be clear that to construct Z_{L^*}, it takes a total of $O(n)+O(n^2)+O(n^4) = O(n^4) = O((\mu+\lambda)^4)$ steps of elementwise operations. Thus, the proof is done. □

5.5.4. Concatenation

In this part, first we will analyze concatenation of URLs, heavily using the results given in Pighizzini's paper "Unary language concatenation and its state complexity" (2000) [11]. Then, with the help of these results, we will analyze closure of concatenation on CUFPFs.

Theorem 5.14. Given $\mu' \geq 0$, $\mu'' \geq 0$, $\lambda' \geq 1$, $\lambda'' \geq 1$, let L_1 and L_2 be URLs accepted by two UDFA D_1 and D_2 of size (λ', μ') (i.e. period λ', transience μ') and (λ'', μ'') (i.e. period λ'', transience μ''), respectively. Then, the concatenation of L_1 and L_2 is accepted by a UDFA of size (λ, μ) where $\lambda = \text{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \text{lcm}(\lambda', \lambda'') - 1$ [11].

Proof. Let our unary alphabet be $\{a\}$. Let X_1, X_2 be the URLs decided by the transient parts of D_1 and D_2, meaning that $X_1 = L_1 \cap \{a^i \mid 0 \leq i < \mu'\}$ and $X_2 = L_2 \cap \{a^i \mid 0 \leq i < \mu''\}$. Let Y_1, Y_2 be the URLs decided by restricting D_1 and D_2 to their periodic parts. Hence, $L_1 = X_1 \cup a^{\mu'}Y_1$ and $L_2 = X_2 \cup a^{\mu''}Y_2$. Then, $L_3 = L_1L_2$ can be written as:

$$\begin{aligned} L_3 &= (X_1 \cup a^{\mu'}Y_1)(X_2 \cup a^{\mu''}Y_2) \\ &= X_1X_2 \cup X_1a^{\mu''}Y_2 \cup X_2a^{\mu'}Y_1 \cup a^{\mu'+\mu''}Y_1Y_2 \end{aligned} \tag{5.2}$$

We can see that L_3 can be represented as a union of four languages. By providing UDFA for every one of these concatenated languages and generating a union of them, we can provide a UDFA deciding L_3. We will start with X_1X_2.

Case of X_1X_2: This is relatively easy. Since X_1 and X_2 is finite, X_1X_2 is also finite. Its longest member is the concatenation of longest members of X_1 and X_2, which can be at most $\mu' - 1 + \mu'' - 1 = \mu' + \mu'' - 2$ symbols long. Thus, a UDFA of size $(1, \mu' + \mu'' - 1)$ can decide X_1X_2.

Case of $X_1a^{\mu''}Y_2$: We will show that there exists a UDFA of size $(\lambda'', \mu' + \mu'' - 1)$ that decides $X_1a^{\mu''}Y_2$. We do this by proving that for any integer $x \geq \mu' + \mu'' - 1$,

$$a^x \in X_1a^{\mu''}Y_2 \quad \text{if and only if} \quad a^{x+\lambda''} \in X_1a^{\mu''}Y_2 \tag{5.3}$$

We know that X_1 is finite and any word in X_1 has length less than μ'. Hence, for an integer $x \geq \mu' + \mu'' - 1$ such that $a^x \in X_1a^{\mu''}Y_2$, there are two integers y_1 and y_2 such that $x = y_1 + y_2$, $a^{y_1} \in a^{\mu''}Y_2$, $a^{y_2} \in X_1$, $y_1 \geq \mu''$, and $y_2 < \mu'$. We also know that $a^{y_1+\lambda''} \in a^{\mu''}Y_2$ and since $a^{y_2} \in X_1$, we can say $a^{y_1+y_2+\lambda''} = v \in X_1a^{\mu''}Y_2$. By similar arguments, one can also prove that, for any $x \geq \mu' + \mu'' - 1$, $a^x \notin X_1a^{\mu''}Y_2$ implies that $a^{x+\lambda''} \notin X_1a^{\mu''}Y_2$. Thus, $a^x \in X_1a^{\mu''}Y_2$ if and only if $a^{x+\lambda''} \in X_1a^{\mu''}Y_2$ for any $x \geq \mu' + \mu'' - 1$. Hence, a UDFA of size $(\lambda'', \mu' + \mu'' - 1)$ can decide $X_1a^{\mu''}Y_2$.

Case of $X_2 a^{\mu'} Y_1$: It is extremely similar to the previous case of $X_1 a^{\mu''} Y_2$. Thus, we can shortly say that there exists a UDFA of size $(\lambda', \mu' + \mu'' - 1)$ that can decide $X_2 a^{\mu'} Y_1$.

Case of $a^{\mu'+\mu''} Y_1 Y_2$: To simplify the case, first we will analyze the concatenated language $Y_1 Y_2$. We will show that $Y_1 Y_2$ can be decided by a UDFA of size $(\gcd(\lambda', \lambda''), \mathrm{lcm}(\lambda', \lambda'') - 1)$. We are doing this by proving that for any integer $z \geq \mathrm{lcm}(\lambda', \lambda'') - 1$,

$$a^z \in Y_1 Y_2 \quad \text{if and only if} \quad a^{z+\gcd(\lambda', \lambda'')} \in Y_1 Y_2 \tag{5.4}$$

but before that we will state a useful lemma (no proof will be given).

Lemma 5.15. Given two positive integers c and d, each number of the form $ci + dj$, with i and j being non-negative integers, is a multiple of $\gcd(c, d)$ and the largest multiple of $\gcd(c, d)$ that cannot be represented as $ci + dj$ is $\mathrm{lcm}(c, d) - (c + d)$ [11].

For an integer $z \geq \mathrm{lcm}(\lambda', \lambda'') - 1$ such that $a^z \in Y_1 Y_2$, there exists non-negative integers i, j, y_1, and y_2 such that $z = y_1 + i\lambda' + y_2 + j\lambda''$ where $a^{y_1 + i\lambda'} \in Y_1$, $a^{y_2 + j\lambda''} \in Y_2$, $y_1 < \lambda'$ and $y_2 < \lambda''$. Since $y_1 + y_2 \leq \lambda' + \lambda'' - 2$ and $z \geq \mathrm{lcm}(\lambda', \lambda'') - 1$,

$$i\lambda' + j\lambda'' = z - y_1 - y_2 \geq \mathrm{lcm}(\lambda', \lambda'') - (\lambda' + \lambda'') + 1 \tag{5.5}$$

One can also see that $i\lambda' + j\lambda''$ is a multiple of $\gcd(\lambda', \lambda'')$. Then, by the previous lemma 5.15, $i\lambda' + j\lambda'' + \gcd(\lambda', \lambda'')$ can be represented as $k_1\lambda' + k_2\lambda''$ for some non-negative integers k_1 and k_2. Thus, we can say the following:

$$z + \gcd(\lambda', \lambda'') = y_1 + y_2 + i\lambda' + j\lambda'' + \gcd(\lambda', \lambda'') = y_1 + k_1\lambda' + y_2 + k_2\lambda'' \tag{5.6}$$

and $a^{y_1 + k_1\lambda'} \in Y_1$ and $a^{y_2 + k_2\lambda''} \in Y_2$. Hence, $a^z \in Y_1 Y_2$ implies $a^{z+\gcd(\lambda', \lambda'')} \in Y_1 Y_2$. By similar arguments, one can also prove that $a^z \notin Y_1 Y_2$ implies $a^{z+\gcd(\lambda', \lambda'')} \notin Y_1 Y_2$. Hence, $a^z \in Y_1 Y_2$ if and only if $a^{z+\gcd(\lambda', \lambda'')} \in Y_1 Y_2$. Thus, we can say that there exists

a UDFA that decides Y_1Y_2 with size $(\gcd(\lambda', \lambda''), \operatorname{lcm}(\lambda', \lambda'') - 1)$.

If there exists a UDFA deciding Y_1Y_2 with size $(\gcd(\lambda', \lambda''), \operatorname{lcm}(\lambda', \lambda'') - 1)$, then surely there exists a UDFA deciding $a^{\mu'+\mu''}Y_1Y_2$ with size $(\gcd(\lambda', \lambda''), \operatorname{lcm}(\lambda', \lambda'') + \mu' + \mu'' - 1)$.

Summary of all cases: In the previous part of this proof, we give the transience and period lengths of four different UDFA that decide our four different URLs X_1X_2, $X_1a^{\mu''}Y_2$, $X_2a^{\mu'}Y_1$, and $a^{\mu'+\mu''}Y_1Y_2$. We know that maximum transience length among those four UDFA is $\operatorname{lcm}(\lambda', \lambda'') + \mu' + \mu'' - 1$. We also know that the least common multiple of those four period lengths is $\operatorname{lcm}(1, \lambda'', \lambda', \gcd(\lambda', \lambda'')) = \operatorname{lcm}(\lambda', \lambda'')$. By theorem 5.8, it is easy to show that union of those four URLs can be decided by a UDFA having a transience of length $\mu' + \mu'' + \operatorname{lcm}(\lambda', \lambda'') - 1$ and a period of length $\operatorname{lcm}(\lambda', \lambda'')$. Hence, the concatenation of L_1 and L_2 is accepted by a UDFA of size (λ, μ) where $\lambda = \operatorname{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \operatorname{lcm}(\lambda', \lambda'') - 1$. □

It is proven that for any $\mu' \geq 2$, $\mu'' \geq 2$, $\lambda' \geq 2$, $\lambda'' \geq 2$ such that $\gcd(\lambda', \lambda'') > 1$, there exists two URLs L_1 and L_2 which are decided by two minimal UDFA D_1 and D_2 of size (λ', μ') and (λ'', μ''), respectively, such that L_1L_2 is decided by a minimal UDFA of size (λ, μ) where $\lambda = \operatorname{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \operatorname{lcm}(\lambda', \lambda'') - 1$ [11]. Hence, the state bound is optimal. Now, we move onto CUFPFs.

Theorem 5.16. CUFPFs are closed under the concatenation operation. Furthermore, say two CUFPFs $Z_{L_1} = \{a, \mu', \lambda', M_1, N_1\}$ and $Z_{L_2} = \{a, \mu'', \lambda'', M_2, N_2\}$ are given for two languages L_1 and L_2 respectively. It takes

$$O(T^2)$$

steps of elementwise operations to construct the concatenation Z_{L_3} of Z_{L_1} and Z_{L_2} where $T = \mu' + \mu'' + 2\operatorname{lcm}(\lambda', \lambda'') - 1$. Z_{L_3} has a transience of $\mu' + \mu'' + \operatorname{lcm}(\lambda', \lambda'') - 1$ and a period of $\operatorname{lcm}(\lambda', \lambda'')$.

Proof. The proof is simple. By theorem 5.14, we know that the concatenated language $L_3 = L_1L_2$ can be decided by a UDFA, say D_3, with a transience of length $\mu' + \mu'' + \operatorname{lcm}(\lambda', \lambda'') - 1$ and a period of length $\operatorname{lcm}(\lambda', \lambda'')$. In that regard, D_3 has a total of $T = \mu' + \mu'' + 2\operatorname{lcm}(\lambda', \lambda'') - 1$ states. Thus, if we can pinpoint the accepting states of such a UDFA, we can explicitly give the deciding UDFA, and so the deciding CUFPF Z_{L_3}.

To know the accepting states of D_3, it is enough for us to know L_3's members which are shorter than T. To do that, we need to generate all members of L_1 and L_2, say a^i and a^j, such that $0 \leq i + j \leq T - 1$. Now, our problem reduces to a very simple problem of finding the sorted cartesian sum of two finite lists of non-negative integers. If we limit i and j such that $i \leq T - 1$ and $j \leq T - 1$, then we can be solve this problem in $O(T^2)$ steps of elementwise addition and comparison operations. The resulting sorted list tells us the accepting states (i.e. if k is an element of the resulting list, then $a^k \in L_3$ and $k + 1 = k^{th}$ state of D_3 is accepting where $0 \leq k \leq T - 1$). Thus, we can explicitly give D_3 and one can easily construct the deciding CUFPF Z_{L_3}. Hence, the proof is done. □

This concludes the fifth chapter.

6. CONCLUSION

6.1. Summary

In this body of work, we study the succinctness properties of various finite automata. First, we study the subject of simulating various finite automata by DFA. We restate the current results regarding the upper and lower bounds of simulation of NFA, 2DFA, 2NFA, AFA, 2AFA, PFA, OQFA, and QFA by DFA. For AFA and OQFA in particular, we give our own definitions and compare our results for simulation bounds to results given before. Second, we give three different families of regular languages and we provide the various minimal automata deciding them. Our contribution is mostly focused on 2DFA and 2NFA that decide those languages; for other cases, we restate the current results proved in various papers. Third, we analyze unary regular languages and unary DFA and create forms called "Unary Finite Periodic Form" and "Compact Unary Finite Periodic Form" to efficiently describe them. We reprove some important theorems for closure properties of unary regular languages, and we introduce algorithms to show the efficient realization of closure properties by using CUFPF.

6.2. Our Contribution

In the third chapter, we give definitions for quantum finite automata that use orthogonal matrices in its computation, namely OQFA. We prove that if a language is recognized by an n-state OQFA with bonded error, then it can be recognized by a DFA with $2^{O(n)}$ states (see theorem 3.17 for the exact bound). This result is parallel to the result given for simulation of simple QFA by DFA, stated in Ambainis and Yakaryilmaz's paper [14].

In the fourth chapter, we prove that to decide B_m, a 2NFA needs at least $\sum_{i=1}^{n} p_i^{k_i}$ where $m = \prod_{i=1}^{n} p_i^{k_i}$ and p_i are prime factors (see theorem 4.8). This result is parallel to the result given in Mereghetti and Pighizzini's paper [26]. In that chapter, we also

prove that at least $m-1$ states are necessary for a 2NFA to decide C_m (see theorem 4.16).

In the fifth chapter, we define minimal CUFPFs and prove that there is only one minimal CUFPF for every unary regular language. We also prove that CUFPFs can be minimized in an efficient manner (i.e. with $O(\mu+\lambda^2)$ steps of basic bitwise operations where μ is transient and λ is period of the CUFPF given, see theorem 5.6).

In the fifth chapter, we also analyze efficient realization of closure properties of URLs using CUFPFs:

- We prove that for any CUFPF $Z_L = \{a, \mu, \lambda, M, N\}$ that describe some URL L, it takes $O(\mu+\lambda)$ steps of basic bitwise operations to construct a CUFPF that describes the complement of L, namely $\bar{L}$.
- We prove that for any CUFPFs $Z_{L_1} = (a, \mu, \lambda, M_1, N_1)$ and $Z_{L_2} = (a, \nu, \gamma, M_2, N_2)$ that describe two URLs L_1 and L_2 respectively, it takes $O(\max(\mu,\nu)+\mathrm{lcm}(\lambda,\gamma))$ steps of basic bitwise operations to construct a CUFPF that describes the union (or, intersection) of L_1 and L_2, namely $L_1 \cup L_2$ (resp. $L_1 \cap L_2$).
- We prove that for any CUFPFs $Z_{L_1} = (a, \mu, \lambda, M_1, N_1)$ and $Z_{L_2} = (a, \nu, \gamma, M_2, N_2)$ that describe two URLs L_1 and L_2 respectively, it takes $O((\mu+\nu+\mathrm{lcm}(\lambda+\gamma))^2)$ steps of basic elementwise operations to construct a CUFPF that describes the concatenation of L_1 and L_2, namely L_1L_2.
- We prove that for any CUFPF $Z_L = (a, \mu, \lambda, M_1, N_1)$ that describes some URL L, it takes $O((\mu+\lambda)^4)$ steps of basic elementwise operations to construct a CUFPF that describes the star of L, namely L^*.

6.3. Open Questions and Future Work

- Regarding the simulations of 2DFA by pDFA (respectively, 2NFA by pDFA), we do not know if there exists a regular language with a much smaller sized alphabet that can be decided by an n-state 2DFA (resp. 2NFA) and a $n(n^n-(n-1)^n)$-state

minimal pDFA (resp. a $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\binom{n}{i}\binom{n}{j}(2^i-1)^j$-state minimal pDFA). Proving or disproving that there exist such regular languages would be an intriguing thing to do.

- Currently, we do not know if there exists a lower bound for the simulation of n-state QFA by DFA that matches with the upper bound $2^{O(n^2)}$ given in theorem 3.19. This is a well-known open problem in quantum computation and clearly needs more attention.
- In the fourth chapter, we study with three different families of regular languages. For those three cases, there is a possibility that we could not have utilized the full power of 2AFA and QFA. Statewise speaking, can they do better against their less sophisticated versions (i.e. 2AFA against AFA, QFA against PFA and OQFA) when deciding those three languages?
- Surely, there are other regular languages that need attention and one can find succinct automata that decides them. For future work, we desire to work on the generalization of A_m and B_m: Finite URLs, periodic URLs, and their complements. There are important connections between them and succinctness of various automata (one can be seen in Mereghetti and Pighizzini's paper [26]), and we want to further investigate those connections.
- In the fourth section of our fifth chapter, we analyze the link between binary primitive words and minimal CUFPFs. It is still not known whether the set of primitive words over a non-unary fixed alphabet is a context-free language or not (see Lischke's paper [29]). To gain further inside about this open problem, it is encouraging to further analyze this connection.

REFERENCES

1. Rabin, M. O. and D. Scott, "Finite Automata and Their Decision Problems", *IBM Journal of Research and Development*, Vol. 3, No. 2, pp. 114–125, April 1959.

2. Shepherdson, J. C., "The Reduction of Two-way Automata to One-way Automata", *IBM Journal of Research and Development*, Vol. 3, No. 2, pp. 198–200, April 1959.

3. Rabin, M. O., "Probabilistic automata", *Information and Control*, Vol. 6, No. 3, pp. 230–245, 1963.

4. Chandra, A. and L. Stockmeyer, "Alternation", *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pp. 98–108, October 1976.

5. Ladner, R., R. Lipton and L. Stockmeyer, "Alternating pushdown automata", *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 92–106, October 1978.

6. Brzozowski, J. and E. Leiss, "On equations for regular languages, finite automata, and sequential networks", *Theoretical Computer Science*, Vol. 10, No. 1, pp. 19–35, 1980.

7. Chandra, A., D. Kozen and L. Stockmeyer, "Alternation", *Journal of the ACM*, Vol. 28, No. 1, pp. 114–133, January 1981.

8. Birget, J. C., "State-complexity of finite-state devices, state compressibility and incompressibility", *Mathematical systems theory*, Vol. 26, No. 3, pp. 237–269, 1993.

9. Birget, J. C., "Two-way automata and length-preserving homomorphisms", *Mathematical Systems Theory*, Vol. 29, No. 3, pp. 191–226, 1996.

10. Ambainis, A. and R. Freivalds, "1-way quantum finite automata: strengths, weaknesses and generalizations", *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)*, pp. 332–341, November 1998.

11. Pighizzini, G., "Unary Language Concatenation and Its State Complexity", *Revised Papers from the 5th International Conference on Implementation and Application of Automata*, CIAA '00, pp. 252–262, Springer-Verlag, London, UK, 2001.

12. Mereghetti, C., B. Palano and G. Pighizzini, "Note on the Succinctness of Deterministic, Nondeterministic, Probabilistic and Quantum Finite Automata", *RAIRO - Theoretical Informatics and Applications*, Vol. 35, No. 5, p. 477–490, 2001.

13. Geffert, V. and A. Okhotin, "Transforming Two-Way Alternating Finite Automata to One-Way Nondeterministic Automata", *Mathematical Foundations of Computer Science 2014*, pp. 291–302, Springer Berlin Heidelberg, 2014.

14. Ambainis, A. and A. Yakaryilmaz, "Automata and Quantum Computing", *CoRR*, Vol. abs/1507.01988.

15. Sipser, M., *Introduction to the Theory of Computation*, Thomson Course Technology, Boston, MA, USA, 2nd edn., 1996.

16. Kapoutsis, C., *Algorithms and Lower Bounds in Finite Automata Size Complexity*, Ph.D. Thesis, Massachusetts Institute of Technology, 2006.

17. Kozen, D., "On parallelism in turing machines", *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pp. 89–97, October 1976.

18. Leiss, E., "Succinct representation of regular languages by boolean automata", *Theoretical Computer Science*, Vol. 13, No. 3, pp. 323–330, 1981.

19. Enderton, H., *A Mathematical Introduction to Logic*, Academic Press, Cambridge, MA, USA, 2nd edn., 2001.

20. Say, A. C. C. and A. Yakaryılmaz, "Quantum Finite Automata: A Modern Introduction", C. S. Calude, R. Freivalds and I. Kazuo (Editors), *Computing with New Resources: Essays Dedicated to Jozef Gruska on the Occasion of His 80th Birthday*, pp. 208–222, Springer International Publishing, Cham, 2014.

21. Dwork, C. and L. Stockmeyer, "A Time Complexity Gap for Two-Way Probabilistic Finite-State Automata", *SIAM Journal on Computing*, Vol. 19, No. 6, pp. 1011–1023, 1990.

22. Gao, Y., N. Moreira, R. Reis and S. Yu, "A Survey on Operational State Complexity", *CoRR*, Vol. abs/1509.03254, 2015.

23. Even, S., L. Selman and Y. Yacobi, "The complexity of promise problems with applications to public-key cryptography", *Information and Control*, Vol. 61, No. 2, pp. 159–173, 1984.

24. Freivalds, R., "On the growth of the number of states in result of determinization of probabilistic finite automata", *Automatic Control and Computer Sciences*, Vol. 1, No. 3, pp. 39–42, 1982.

25. Balodis, K., "Counting with Probabilistic and Ultrametric Finite Automata", C. S. Calude, R. Freivalds and I. Kazuo (Editors), *Computing with New Resources: Essays Dedicated to Jozef Gruska on the Occasion of His 80th Birthday*, pp. 3–16, Springer International Publishing, Cham, 2014.

26. Mereghetti, C. and G. Pighizzini, "Two-Way Automata Simulations and Unary Languages", *Journal of Automata, Languages and Combinatorics*, Vol. 5, No. 3, pp. 287–300, 2000.

27. Ambainis, A. and N. Nahimovs, "Improved Constructions of Quantum Automata", Y. Kawano and M. Mosca (Editors), *Theory of Quantum Computation, Communication, and Cryptography: Third Workshop, TQC 2008 Tokyo, Japan, January 30 - February 1, 2008. Revised Selected Papers*, pp. 47–56, Springer Berlin Heidelberg,

Berlin, Heidelberg, 2008.

28. Rasscevskis, Z., "The complexity of probabilistic vs deterministic finite automata", Unpublished paper.

29. Lischke, G., "Primitive words and roots of words", *Acta Universitatis Sapientiae, Informatica*, Vol. 3, No. 1, pp. 5–34, 2011.

30. Shyr, H. J. and G. Thierrin, "Disjunctive languages and codes", M. Karpiński (Editor), *Fundamentals of Computation Theory: Proceedings of the 1977 International FCT-Conference, Poznań-Kórnik, Poland September 19–23, 1977*, pp. 171–176, Springer Berlin Heidelberg, Berlin, Heidelberg, 1977.

31. Hopcroft, J., *An N Log N Algorithm for Minimizing States in a Finite Automaton*, Tech. rep., Stanford, CA, USA, 1971.

32. Shallit, J., "State Complexity and Jacobsthal's Function", S. Yu and A. Păun (Editors), *Implementation and Application of Automata: 5th International Conference, CIAA 2000 London, Ontario, Canada, July 24–25, 2000 Revised Papers*, pp. 272–278, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

33. Yu, S., Q. Zhuang and K. Salomaa, "The State Complexities of Some Basic Operations on Regular Languages", *Theoretical Computer Science*, Vol. 125, No. 2, pp. 315–328, March 1994.