THE EMPTINESS PROBLEM FOR EXTENDED REGULAR EXPRESSIONS

By

MIN CAI

Bachelor of Science

Shantou University

Shantou, China

1997

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2003

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my major advisor, Dr. H. K. Dai for his intelligent supervision, constructive guidance, inspiration and encouragement. My sincere appreciation extends to my other committee members Dr. Hedrick and Dr. Chandler, whose guidance, assistance and support are also invaluable.

More over, I wish to express my sincere gratitude to Dr. H. G. W. Burchard, who provided suggestions and assistance for this study.

I would also like to give my special appreciation to my parents in Hong Kong, my two elder brothers, Ben Tsoi, and Yingchao Cai whose support, encouragement and love go through my study in US.

Contents

LIST OF FIGURES

Chapter 1

Preliminaries

1.1 Problems, Algorithms, and Complexity

A problem is a general question to be answered, and it has several parameters. To describe a problem, we describe all its parameters, and state what properties the answer is required to satisfy. Given a problem, if we specify particular values for all its parameters, then it is an instance of the problem. The input length for an instance of a problem is defined to be the number of symbols in the description of a problem instance. If a problem Π has only two possible solutions, either "yes" or "No", then it is a decision problem. Such a problem Π consists of an instance set D_Π and a subset $Y_\Pi \subseteq D_\Pi$ of yes-instances.

Algorithms are procedures for solving problems, they are general, step-by-step. If an algorithm can be used for any instance of a problem and can always solve that instance, then we say that this algorithm solve the problem. In general, we are interested in finding the most "efficient" algorithm for solving a problem. Time and space are usually two factors determining whether or not an algorithm is efficient.

Complexity theories provide mechanisms to classify combinatorial problems and measure computational resources that are necessary to solve them. Two most important and common measures in the computation are time and space complexities. The time complexity is the number of steps a program takes to execute, and the space complexity is the amount of storage used in the computation.

1.2 The Time Complexity

1.2.1 Deterministic Computation and the Class P

Deterministic Turing Machine is a particular model for computation. It is defined as follows:

Definition 1 (Deterministic Turing Machine) *[10]*

A deterministic Turing machine is a system

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, q_{accept}, q_{reject})$, *where*

Q *is the finite set of states,*

Γ *is the finite tape alphabet,*

$B \in \Gamma$ *is the blank symbol,*

Σ *is the input alphabet,* $\Sigma \subset \Gamma - \{B\}$,

$q_0 \in Q$ *is the initial state,*

q_{accept} *is the accepting state,*

q_{reject} *is the rejecting state, and*

δ *is the transition function,* $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.

P is the class of all languages accepted by a deterministic Turing machine program that runs in polynomial time in the input length. A polynomial time algorithm is the polynomial time deterministic Turing machine program.

1.2.2 Nondeterministic Computation and the Class NP

Informally the class NP can be defined in terms of a nondeterministic algorithm. Such an algorithm consists of two separate stages: guessing stage and checking stage. Given a problem instance I, the guessing stage "guesses" some structure S. Then we take I and S as inputs of checking stage and compute it in a normal deterministic manner. A nondeterministic algorithm "solves" a decision problem if, for all instants of the decision problem, there exists some structure S that, when guessed for input I, will lead the checking stage to respond "yes" if and only if I is a yes-instance.

A nondeterministic algorithm is said to solve a decision problem in "polynomial time" if, for every yes-instance, there is some guess S that leads the deterministic checking stage to respond "yes" in the polynomial time in the input length. The class NP is defined informally to be the class of all decision problems that can be solved by polynomial time nondeterministic algorithms.

A nondeterministic Turing machine is the same as a Turing machine, except that the transition function has the following form:

$\delta : (Q - \{q_{accept}\}) \times \Gamma \rightarrow \wp\,(Q \times \Gamma \times \{L, R\})$, where $\wp(A)$ is the power set of a set A.

The formal counterpart of a nondeterministic algorithm is a program for a nondeterministic Turing machine.

The language recognized by a nondeterministic Turing machine is the set of all input strings accepted by the nondeterministic Turing machine. The time required by a nondeterministic Turing machine to accept a string is defined to be the minimum, over all accepting computations of the nondeterministic Turing machine, of the number of steps occurring in the guessing and checking stages up until the halt state is entered.

NP is the class of all languages accepted by a nondeterministic Turing machine program that runs in polynomial time in the input length.

We usually envision a nondeterministic algorithm as guessing a structure that in some way depends on the given instance. The guessing module of a nondeterministic Turing machine disregards the given input. However, we always design out a nondeterministic Turing machine program so that the checking stage begins by checking whether or not the guessed string corresponds to an appropriate guess for the given input. If not, the program can halt immediately.

1.2.3 The Relationship between P and NP

We observe that every deterministic algorithm can be used as the checking stage of a nondeterministic algorithm. From this, we can conclude that every decision problem that can be solved by a polynomial time deterministic algorithm can be also solved by a polynomial time nondeterministic algorithm. That implies that $P \subseteq NP$. There are many reasons to believe that this inclusion is proper, which means that P is not equal to NP [8]. It is not surprising that polynomial time nondeterministic algorithms are more powerful than polynomial time deterministic algorithms, even though it has not been proved as yet. Therefore, it looks reasonable to assume that $P \neq NP$.

1.2.4 The Structure of NP

If $P \neq NP$, then the distinction between P and NP - P is meaningful and important. There is no hope of showing that any problem is in NP - P until we can prove that $P \neq NP$.

Definition 2 (Polynomial Transformation) *[8]*

A polynomial transformation from one language $L_1 \subseteq \Sigma_1^*$ to another language $L_2 \subseteq \Sigma_2^*$ is a function $f : L_1 \longmapsto L_2$ that satisfies two conditions:

1. *There exists a polynomial time deterministic Turing machine program computing f, and*

2. *For all $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$.*

It is obvious that the "polynomial transformability" relation is reflexive and transitive, but not symmetric. A language L_1 is defined to be NP-complete if L_1 is in NP and, for any language L_2 that is in NP, there exists a polynomial transformation from L_2 to L_1. The NP-complete theory focuses on proving results of weaker form "if P $\neq$ NP, then a decision problem is in NP - P". NPC is made up of all NP-complete languages. The "polynomial transformability" relation imposes a partial order on the equivalence classes of languages (or decision problems). The class P is the "least" equivalence class under this partial order. The class of NP-complete problems contains the "hardest" languages (or decision problems) in NP. If any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time. Therefore, if P $\neq$ NP, then any NP-complete problem is in NP - P. Moreover, any NP-complete problem is in P if and only if P = NP.

Problems in NP are considered to be in NPI if they have not yet been proved either in P or in NPC. Since it has been showed that there are some problems in NPI [8], we can conclude that if P $\neq$ NP, then there exist some problems in NP neither solvable in polynomial time nor NP-complete. That is, there exists some problems in NP but not in P or NPC. The class of all languages that are not in P or NPC but in NP is the class NPI.

Assuming that P $\neq$ NP, the NP class consists of three parts: P, NPC, and NPI. Their "difficulty" levels from the least difficult to the most difficult are P, NPI, NPC.

1.2.5 Some Problems in NP

As we mentioned before, the P class includes all languages accepted by deterministic Turing machines in polynomial time, without any regard to the degree of the polynomial. *Integer Divisibility by Four [8]*, *Primes* and its complement problem *Composite Numbers* have been proved to be in P [9].

Integer Divisibility by Four

Instance: Given an integer $n \geq 1$.

Question: Is there an integer $m \geq 1$, such that $n \,/\, m = 4$ and $n \; mod \; m = 0$?

Primes

Instance: Given an integer $k \geq 1$.

Question: Is k a prime?

Composite Numbers

Instance: Given an integer $k \geq 1$.

Question: Are there two integers m, $n \geq 2$, such that $mn = k$?

The NPC class includes many problems that are natural and have been solved efficiently [8]. Some problems such as the *Vertex Cover* problem, the *Hamilton Circuit* problem, and the *Clique* problem have been proved to be NP-complete [7]. For all of these problems we can find exponential algorithms, but so far no polynomial time algorithm has been found to solve any of the NP-complete problems.

Vertex Cover

Instance: Given a graph $G = (V, E)$ and an integer k, where $1 \leq k \leq |V|$.

Question: Is there a set $V' \subseteq V$, such that $|V'| \leq k$ and, $\forall \{u, v\} \in E$, $\exists$ at least one u or v in V' ?

Hamiltonian Circuit

Instance: Given a graph $G = (V, E)$.

Question: Is there an ordering of vertices of G, $\langle v_1, v_2, \ldots, v_n \rangle$, where $n = |V|$, such that $e_i \in E$, $\forall\ 1 \leq i \leq n$, where $e_i = \{v_i, v_{i+1}\}$, $\forall\ 1 \leq i \leq n - 1$, and $e_n = \{v_n, v_1\}$?

Clique

Instance: Given a graph $G = (V, E)$ and an integer j, where $1 \leq j \leq |V|$.

Question: Is there a set $V' \subseteq V$ such that $|V'| \geq j$ and for any two vertices v_i, $v_j \in V'$, $\{v_i, v_j\} \in E$?

NPI consists of problems in NP which have not yet been proved either in P or in NPC. The NPI class is not empty if P $\neq$ NP. *Graph Isomorphism*, and *Linear Programming* are some examples of problems in NPI [8].

Graph Isomorphism

Instance: Given two graphs $G = (V, E)$ and $G' = (V, E')$.

Question: Is there an injective function $f: V \rightarrow V$, such that $\{u, v\} \in E$ if and only if $\{f(u), f(v)\} \in E'$?

Linear Programming

Instance: Given an integer B and three integer vectors $V_i = (v_i[1], v_i[2], \ldots, v_i[n])$, $D = (d_1, d_2, \ldots, d_m)$, and $C = (c_1, c_2, \ldots, c_n)$, where $1 \leq i \leq m$.

Question: Is there a rational vector $X = (x_1, x_2, \ldots, x_n)$ such that $V_i \cdot X \leq d_i$ and $C \cdot X \geq B$, where $1 \leq i \leq m$?

1.2.6 The Class Co-NP

The class co-NP is the set of all languages whose complement is in NP. It is defined as follows:

Definition 3 (co-NP) *[8]*

$$co - NP = \{\Sigma^* \text{ - } L \mid L \textit{ is a language over the alphabet } \Sigma \textit{ and } L \in NP.\}$$

Many problems in co-NP seem not to be in NP, which means NP $\neq$ co-NP. The class P is closed under complementation, so NP $\neq$ co-NP implies P $\neq$ NP, although P $\neq$ NP does not imply NP $\neq$ co-NP. Nevertheless, there exists a link between the NP-complete problems and the conjecture that NP $\neq$ co-NP. This link is that if there is an NP-complete problem whose complement is in NP, then NP = co-NP [8]. From this, we can conclude that a problem whose complement is in NP can not be in NPC unless NP = co-NP.

1.2.7 Exponential Time

There are some decision problems only solvable by a Turing machine in exponential time. The set of all decision problems that can be solved by a deterministic [respectively nondeterministic] Turing machine in O($2^{p(n)}$) time, where n is the input length, $p(n)$ is a polynomial function of n, is the class EXPTIME [respectively NEXPTIME]. For all polynomial functions $q(n)$, $m^{q(n)} = 2^{p(n)}$, where m is any positive integer, and $p(n)$ is a polynomial function of n, the class EXPTIME [respectively NEXPTIME] is also the set of all decision problems solved by a deterministic [respectively nondeterministic] Turing machine in O($m^{q(n)}$) time, where m is any positive integer, and $q(n)$ is a polynomial function of n. We believe that there exists some decision problems that are beyond the class EXPTIME. That means, those problem can only be solved by a deterministic or nondeterministic Turing machine in O($2^{p'(n)}$) time, where $p'(n)$ is an exponential function of n.

The class EXPTIME-complete is also a set of decision problems. A decision problem is in EXPTIME-complete [respectively NEXPTIME-complete] if it is in EXPTIME [respectively NEXPTIME], and every problem in EXPTIME [respectively NEXPTIME] has a polynomial transformation to it. EXPTIME-complete might be thought of as the hardest problem in EXPTIME. EXPTIME is a strict superset of NP-Complete, NP, and P.

One example of EXPTIME-complete problems is the *Chess* problem [10].

Chess

Instance: Given a chess or go position.

Question: Can the first player force a win?

For the *Chess* problem, actually, the games have to be generalized by playing them on an $n \times n$ board instead of the usual board with fixed size. Since EXPTIME-complete is defined by asymptotic behavior as the problem size grows without bound.

1.2.8 The Polynomial Hierarchy

For two sets A and B, we can write a program that is an acceptor for A and allow it to make subroutine calls of the form "$y \in B$" . These calls return true if the Boolean test is true and return false otherwise. Such a program is called a reduction procedure and the set B is called an oracle set. An oracle Turing machine is a standard Turing machine with an additional oracle tape and three special states: Q, YES and NO. When the Turing machine is in state Q, if the word currently written on the oracle tape is in the oracle set, then the next state is YES, otherwise, the next state is NO. A Turing reduction from one oracle set A to another oracle set B is an oracle Turing machine M whose oracle is B such that M accepts A and M halts on every input [11].

The polynomial hierarchy is a useful tool to classify and measure the complexity of combinatorial problems. A class P^Y [respectively NP^Y] is the set of languages from which there is a polynomial time deterministic [respectively nondeterministic] Turing reduction to a language in Y. A class $co-Y$ is the set of languages whose complement is in Y. The polynomial hierarchy is defined inductively as follows [12]:

$$\Sigma_0^p = \Pi_0^p = \Delta_0^p = P \tag{1.1}$$

For all $k \geq 0$

$$\Delta_{k+1}^{p} = P^{\Sigma_k^p} \tag{1.2}$$

$$\Sigma_{k+1}^{p} = NP^{\Sigma_k^p} \tag{1.3}$$

$$\Pi_{k+1}^{p} = co - \Sigma_{k+1}^{p} \tag{1.4}$$

The process of inductively defining new classes can be extended infinitely and it creates classes of greater and greater apparent difficulty. To check if a problem is in the hierarchy at all or not, it is useful to show it is in a particular class directly rather than apply the inductive definitions. If we can show a "hardest" problem in Σ_k^p, then it must be in Σ_k^p - Σ_{k-1}^p if the two classes are not equal.

The polynomial hierarchy extends the classes P and NP. Nevertheless, under the assumption P $\neq$ NP, the polynomial hierarchy remains of theoretical interest. It is not known whether any of the classes are distinct or whether there exists infinitely many classes in the polynomial hierarchy so far.

1.3 The Space Complexity

1.3.1 The Class PSPACE

What we have introduced above is just on the time complexity. In practice, the space complexity is also important. In a Turing machine computation, the time complexity is the number of steps taken before a halt state is entered. The space complexity is the number of distinct tape squares visited by the read-write head. The number of tape squares visited is less than or equal to the number of steps in the computation. It follows that all decision problems in polynomial time can be solved in polynomial space, however, there still exists some decision problems in polynomial space that cannot be solved in polynomial time.

The class PSPACE [8] is the set of languages that are recognizable by polynomial space bounded deterministic Turing machines that halt on every input. There exist some problems solvable in PSPACE that appear to be "harder" than problems in P or NP.

PSPACE is a class beyond the polynomial hierarchy. A language L_1 is PSPACE-complete (with respect to polynomial transformability) if L_1 is in PSPACE and, for any language L_2 that is in PSPACE, there exists a polynomial transformation from L_2 to L_1. From this, we can conclude that if L_1 is PSPACE-complete, then L_1 is in P[respectively NP] if and only if P[respectively NP] = PSPACE. *Quantified Boolean Formulas* and *Linear Space Acceptance* are PSPACE-complete.

Quantified Boolean Formulas

Instance: Given a formula $F = (Q_1x_1)(Q_2x_2) \dots (Q_nx_n)E$, where E is a Boolean expression over x_i, and $Q_i \in \{\exists, \forall\}$, for all $1 \leq i \leq n$.

Question: Is F true?

Linear Space Acceptance

Instance: Given a linear bounded deterministic Turing machine M and a finite string x that is over the input alphabet of M.

Question: Does the Turing machine M accept the string x?

1.3.2 The Class NPSPACE

The class of NPSPACE consists of those languages that can be recognized by a nondeterministic Turing machine in polynomial space bounded. How to deal with the space used by the "guess" in a nondeterministic Turing machine? In fact, for many computations, it is not necessary to remember all the symbols once they have been read. A nondeterministic Turing machine that is used to measure space can be viewed as an additional device from which the program can always request the immediately following symbol of the guess without using any other space. The program records the symbol that is needed later on its tape and use "space" for it only if the program wants to remember the symbol. Defining the class NPSPACE to be the set of languages recognized by programs for this additional device in polynomially bounded space in its accepting computation, then we can ask a question: Is PSPACE equal to NPSPACE? Savitch has implied that the answer is "yes" [15]. This will follow that PSPACE-completeness is the strongest type of completeness result we have introduced above.

1.3.3 The Class DLOGSPACE

Since an input string with length n takes up n tapes squares by itself, any deterministic Turing machine seems taking at least linear space. However, it is different between the space required by the input string and the additional space in which the computation is carried out. In fact, it is possible to use less than linear space for a computation. This is the class DLOGSPACE. The class DLOGSPACE is the class in which all languages can be recognized by a deterministic Turing machine in a space that is only logarithmic in the input length. It is within P and NP.

There are some nontrivial problems solvable in logarithmic space. It has been shown that DLOGSPACE $\neq$ P [15]. Many problems in P look to require more than logarithmic space. Moreover, both P = DLOGSPACE and P = PSPACE can not be held.

Since polynomial transformation can not make distinctions within P, let us introduce log-space transformation.

Definition 4 (Log-space Transformation). *[8]*

A log-space transformation from one language $L_1 \subseteq \Sigma_1^*$ to another language $L_2 \subseteq \Sigma_2^*$ is a function $f : L_1 \longmapsto L_2$ that satisfies two conditions:

1. *f can be computed by a deterministic Turing machine program using space bounded by $\lceil \log n + 1 \rceil$, where n is the input length, and*

2. *For all $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$.*

A language L_1 is log-space complete for P if L_1 is in P and, for any language L_2 that is in P, there exists a log-space transformation from L_2 to L_1. If there exists a log-space transformation from one problem A in P to another problem B in P, then we can conclude that problem A is also log-space complete for P. The *Path System Accessibility* problem has been proved to be log-space complete for P [5].

Path System Accessibility

Instance: Given a finite set X, a relation $R \subseteq X \times X \times X$, and two sets S and T of "source" and "terminal" nodes, where $S, T \subseteq X$.

Question: Is there an "accessible" terminal node, where a node $x \in X$ is accessible if $x \in S$ or if there are accessible nodes y and z such that $\langle x, y, z \rangle \in R$?

Log-space transformation is not only used to prove log-space completeness for P. Most transformations used to prove NP-completeness and PSPACE-completeness are also log-space transformations. The set of all languages that are log-space complete for NP is at least a large subset of NPC, while we can not conclude that all languages that are log-space complete are PSPACE-complete.

1.3.4 The Class NLOGSPACE

The log-space transformation can be used to address another question of determinism versus nondeterminism. Similar to the class DLOGSPACE, the class NLOGSPACE is the class of all languages in which all languages can be recognized by a nondeterministic Turing machine in space-bounded of logarithmic in the input length. It has been proved that there are some languages in NLOGSPACE but not in DLOGSPACE [16], therefore, DLOGSPACE $\neq$ NLOGSPACE. Like DLOGSPACE $\subseteq$ P, NLOGSPACE $\subseteq$ P.

1.3.5 Exponential Space

Similar to the time complexity, there are some decision problems unsolvable by a Turing machine in polynomial space. The class EXPSPACE [respectively NEXPSPACE] is the set of decision problems which are solved by a deterministic [respectively nondeterministic] Turing machine in $O(2^{p(n)})$ space, where n is the input length, and $p(n)$ is a polynomial function of n. That is, the class EXPSPACE [respectively NEXPSPACE] is the set of all decision problems which are solved by a deterministic [respectively nondeterministic] Turing machine in $O(m^{q(n)})$ space, where m is any positive integer, and $q(n)$ is a polynomial function of n. The class EXPSPACE-complete is also a set of decision problems. A decision problem is EXPSPACE-complete [respectively NEXPSPACE-complete] if it is in EXPSPACE [respectively NEXPSPACE], and every problem in EXPSPACE [respectively NEXPSPACE] has a polynomial transformation to it. EXPSPACE-complete might be thought of as the hardest problems in EXPSPACE. EXPSPACE is a superset of EXPTIME, PSPACE, NP-complete, NP, and P. We believe that there exists some decision problems that are beyond EXPSPACE and those problems can only be solved by a deterministic or nondeterministic Turing machine in $O(2^{p'(n)})$ space, where $p'(n)$ is an exponential function of n.

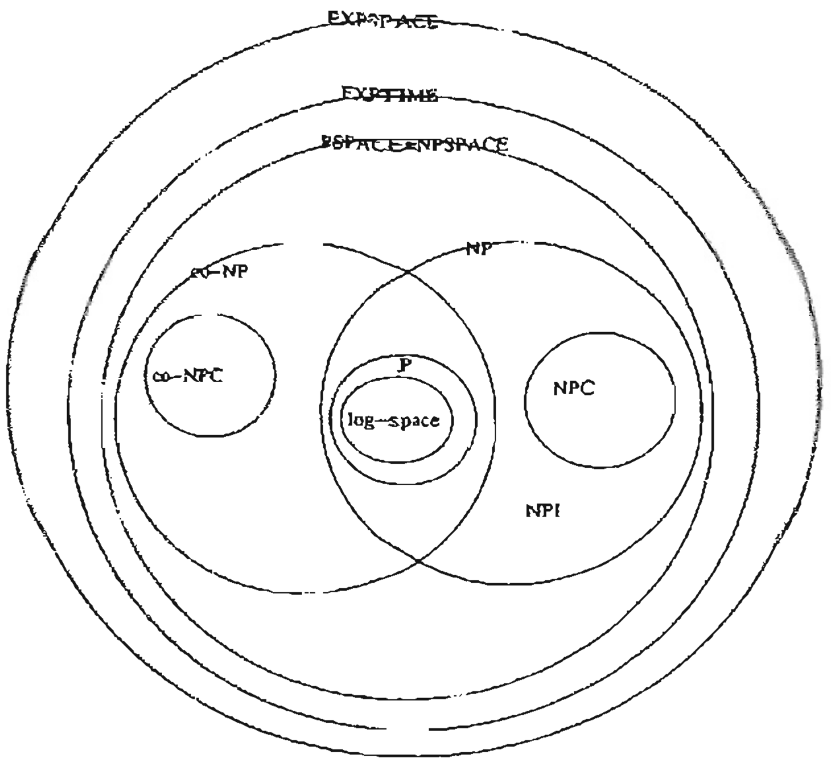

Figure 1.1: The world of complexity classes.

1.4 Relations of the Standard Complexity Classes

Figure 1.1 shows the relationship of complexity classes we introduced above.

Chapter 2

Intractable Problems for Extended Regular Expressions

2.1 Intractability

Different algorithms process different time and space complexities, and which are "efficient enough" and which are "too inefficient" will always depend on the situation at hand. Fortunately, the distinction between polynomial time algorithms and exponential time algorithms offers considerable insight into these matter. Polynomial time algorithms are considered as "good" algorithms, whereas exponential time algorithms are not "good" algorithms.

A problem has not been "well-solved" until a polynomial time algorithm can be found for it. Hence, a problem is said to be intractable if there is no polynomial time algorithm that can solve it. That means, all algorithms to solve an intractable problem require at least exponential time. If there exists a polynomial time bounded algorithm to solve a problem, then this problem is tractable. Although an exponential time algorithm may be faster than a polynomial time algorithm for a problem instance with some limited input length, such kind of problems are quite rare in practice. Therefore, it is appropriate to define intractability as above.

There are two reasons for the intractability of a problem: one is that the problem is so difficult that an exponential time algorithm is needed to solve it, the other is that the solution itself is so extensive that it can not be described with an expression with length bounded by a polynomial function of the input length. To show a problem is intractable, we need to show it is beyond the class P. Since several complexity classes have been known to contain intractable sets, one approach to prove a problem is intractable is by proving that the problem is complete for a complexity class that is known to contain intractable problems.

To prove a particular problem is NP-complete or PSPACE-complete, we show how to express an arbitrary problem in NP or PSPACE in terms of the particular problem. Essentially the technique of proof is simulation. However, so far nobody can find a problem in NP or PSPACE but be proved not in P. To show a problem is not in P, we need to show there exists at least one language not accepted by any deterministic Turing machine in polynomial time. The diagonalization technique is usually used.

If a deterministic Turing machine whose input is a string of n 1's halts after it takes exactly $F(n)$ steps, then we call $F(n)$ a time constructible function. For the time complexity, even though P $\neq$ NP is still an open question, we observed that for two given time constructible functions T_1 and T_2, if T_1 grows "faster" than T_2, then there is a language accepted by a deterministic Turing machine of time complexity T_2 but by no deterministic Turing machine of time complexity T_1.

For the space complexity, we have observed that any multitape deterministic Turing machine has an equivalent one-tape deterministic Turing machine, and both deterministic Turing machines have the same space complexity. If a deterministic Turing machine whose input is a string of n 1's halts after its read-write head has visited exactly $F(n)$ tape squares, then we call $F(n)$ a space constructible function. Furthermore, it has been proved that for two given space constructible functions S_1 and S_2, if S_1 grows "faster" than S_2, then there is a language accepted by a deterministic Turing machine of space complexity S_2 but by no deterministic Turing machine of space complexity S_1.

2.2 Intractable Problems for Extended Regular Expressions

2.2.1 Regular Expressions

A regular expression is recursively defined as follows:

Definition 5 (Regular Expression) *[10]*

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote

1. $\emptyset$ is a regular expression and denotes the empty set.

2. ϵ is a regular expression and denotes the set $\{\epsilon\}$.

3. For each a in Σ, a is a regular expression and denotes the set $\{a\}$.

4. If r and s are regular expressions denoting the languages R and S, respectiv then $(r+s)$, (rs), and (r^*) are regular expressions that denote the sets $R \cup$ RS, and R^*, respectively.

Some intractable problems concerning regular expressions are the followings:

Regular Expression Inequivalence

Instance: Given two regular expressions R_1 and R_2 over an alphabet Σ.

Question: Does the language denoted by R_1 differ from the language denoted by

Let us introduce some related definitions before giving the complexity for the *Regu Expression Inequivalence* problem.

Let X be a finite alphabet, a language $L \subseteq X^*$ is bounded if there exist words w_2, ... $w_n \in X^*$ such that $L \subseteq w_1^* w_2^* ... w_n^*$. If L is not bounded, then we call unbounded language [4]. Star height of a regular expression is a limited nesting de of Kleene stars in the regular expression. If the star height of a regular expressic equal to 0, then it is star free [3].

The complexity for the *Regular Expression Inequivalence* problem depends on R_1 and R_2.

- If R_2 is a fixed expression and denoting an "unbounded" language, then the *Regular Expression Inequivalence* problem is PSPACE-complete,
- If R_2 is a fixed expression and denoting an infinite "bounded" language, then the *Regular Expression Inequivalence* problem is NP-complete,
- If R_2 is a fixed expression and denoting an finite language, then the *Regular Expression Inequivalence* problem is solvable in polynomial time,
- If both R_1 and R_2 have star height k and k is a fixed number which is greater than 0, then the *Regular Expression Inequivalence* problem is PSPACE-complete,
- If both R_1 and R_2 are star free, then the *Regular Expression Inequivalence* problem is NP-complete,
- If one or both of R_1 and R_2 denote bounded languages or the size of Σ is exactly equal to 1, then the *Regular Expression Inequivalence* problem is NP-complete [6],
- If the regular expressions are limited to four operators: union, concatenation, the Kleene star, and squaring (two copies of an expression), then the *Regular Expression Inequivalence* problem is EXPSPACE-complete, and
- If the Kleene star is left out, then the *Regular Expression Inequivalence* problem is NEXPTIME-complete.

For the *Regular Expression Inequivalence* problem, in the case of the size of Σ is

and R_2 is Σ^*, we can get the following problem.

Regular Expression Non-universality

Instance: Given a regular expression R over a finite alphabet Σ.

Question: Does the language denoted by R differ from Σ^*?

The *Regular Expression Non-universality* problem is PSPACE-complete if the alph

bet is {0,1}. But if we allow the abbreviation $(a)^2$ denoting a · a, then the probl

will become intractable and has been shown to have exponential space complexity [

Emptiness-of-complement of Regular Expressions

Instance: Given a regular expression R over the alphabet Σ.

Question: Is the complement of the language denoted by R empty?

By constructing of a PSPACE nondeterministic Turing machine, the *Emptiness-*

complement of Regular Expressions problem can be proved to be PSPACE-compl

[1]. Since the PSPACE class is known to include the set of intractable problems, t

Emptiness-of-complement of Regular Expressions problem is intractable.

2.2.2 Semiextended Regular Expressions

Since the class of regular set is closed under intersection, it does not increase the cl

of sets described by adding the intersection operator to "ordinary" regular expr

sions. The semiextended regular expression allows the intersection operator to be regular expressions. An intractable problem of semiextended regular expressions

the *Emptiness-of-complement of Semiextended Regular Expressions* problem.

Emptiness-of-complement of Semiextended Regular Expressions

Instance: Given a semiextended regular expression R over the alphabet Σ.

Question: Is the complement of the language denoted by R empty?

The *Emptiness-of-complement of Semiextended Regular Expressions* problem has be proved to not in NPSPACE, by the relationship between PSPACE and NP, we kn that under the assumption P $\neq$ NP, we have NP is a subset of PSPACE, and P is subset of NP, hence the *Emptiness-of-complement of Semiextended Regular Expr sions* problem is not to be in P, NP, or PSPACE, that is, there is no polynomial-tin bounded or polynomial-space-bounded algorithm for the *Emptiness-of-complement Semiextended Regular Expressions* problem. Furthermore, it has been shown that a algorithm to solve the *Emptiness-of-complement of Semiextended Regular Expressio* problem requires more than $c^{'}c^{\sqrt{n/\log n}}$ time and space, where $c^{'} \geq 1$ and $c \geq 2$, n the length of the semiextended regular expression[1].

One equivalent problem of the *Emptiness-of-complement of Semiextended Regu Expressions* problem is that whether a given semiextended regular expression denot all strings over its alphabet, so any algorithm to decide whether a semiextend regular expression denotes all strings over its alphabet requires at least PSPAC complexity, particularly, it has been proved its space and time complexity are least $c^{'}c^{\sqrt{n/\log n}}$, where $c^{'} \geq 1$ and $c \geq 2$, n is the length of the semiextended regul expression. This leads us to consider the time and space complexities of the equival

problem for semiextended regular expressions. Fortunately, it has been proved t

the equivalent problem for semiextended regular expression requires $c' c^{\sqrt{n/\log n}}$, wh

$c' \geq 1$ and $c \geq 2$, and an infinity of n's.

2.2.3 Extended Regular Expressions

Similar to the intersection operator, the class of regular set is closed under the co

plementation operator. Extended regular expressions allow intersection and comp

mentation operators in regular expressions. Formally, it is defined as the followin

Definition 6 (Extended Regular Expression) *[1]*

An extended regular expression over an alphabet Σ is defined as follows:

1. *ϵ, $\emptyset$ and α for each in Σ, are extended regular expressions denoting $\{\epsilon\}$,*
   empty set, and $\{\alpha\}$, respectively.

2. *If R_1 and R_2 are extended regular expressions denoting the languages L_1*
   L_2, respectively, then $(R_1 + R_2)$, $(R_1 \cdot R_2)$,$(R_1{}^*)$, $(R_1 \cap R_2)$, and $(\neg R_1)$
   extended regular expressions, denoting $L_1 \cup L_2$, $L_1 L_2$, $L_1{}^*$, $L_1 \cap L_2$, and Σ
   L_1, respectively.

Adding the intersection and complementation operators will make the length of

expressions denoting certain regular languages shorter, but some problems for

tended regular expressions require more time than regular expressions.

One intractable problem in extended regular expressions is the *Emptiness Proble*

for Extended Regular Expressions.

Emptiness Problem for Extended Regular Expressions

Instance: Given a extended regular expression R over the alphabet Σ.

Question: Is the language denoted by R empty?

The *Emptiness Problem for Extended Regular Expressions* is harder than the *Emptin*
of-complement of Regular Expressions problem or the *Emptiness-of-complement*
Semiextended Regular Expressions problem.

If a function $g(m, n)$ is defined as:

$$g(m,n) = \begin{cases} n & \text{for } m = 0 \\ 2^{g(m-1,n)} & \text{for } m > 0 \end{cases} \quad (2.$$

Then a function is elementary if it is bounded above for all but a finite set of n's
g(m_0, n) for some fixed m_0.

Based on the elementary function and impossibility of construction of determinist
Turing machines for extended regular expressions, it has been shown that there
no elementary function $S(n)$ for which the *Emptiness Problem for Extended Regul*
Expressions is of space complexity with $S(n)$ [1]. Moreover, there is no $S(n)$-spac
bounded or $S(n)$-time-bounded deterministic Turing machine to decide whether t
language denoted by a extended regular expression is empty or not.

It has been proved that the *Emptiness Problem for Extended Regular Expressions* beyond PSPACE and any algorithm to solve the *Emptiness Problem for Extend Regular Expressions* requires at least exponential time and space complexity. [1]. particular, the *Emptiness Problem for Extended Regular Expressions* requires at lea $g(m,n)$ time complexity, where m is any finite number. Therefore, the *Emptine Problem for Extended Regular Expressions* is intractable.

Chapter 3

Emptiness Problem for Extended Regular Expressions

3.1 An Algorithm for Solving the Emptiness Prob lem for Extended Regular Expressions

In this thesis we present an algorithm to solve the emptiness problem for extende regular expressions. By analyzing the complexity of our algorithm, we verify th the emptiness problem for extended regular expressions requires at least exponenti time and space. Our algorithm consists of two parts: one is constructions of fini automata, the other is the reachability algorithm.

3.1.1 Constructions of Finite Automata

Finite automata are useful tools to present regular expressions. They are defined

the followings:

Definition 7 (Finite Automaton) *[10]*

A finite automaton, or finite-state machine (abbreviated FA) is a 5-tuple (Q,

q_0, δ, A), where

- *Q is a finite set (whose elements we will think of as states),*
- *Σ is a finite alphabet of input symbols,*
- *$q_0 \in Q$ (the initial state),*
- *$A \subseteq Q$ (the set of accepting states), and*
- *δ is a function from $Q \times \Sigma$ to Q (the transition function).*

For any element $q \in Q$ and any symbol $a \in \Sigma$, we interpret $\delta(q, a)$ as the state

which the FA moves, if it is in state q and receives the input a.

Definition 8 (Nondeterministic Finite Automaton) *[10]*

A nondeterministic finite automaton, (abbreviated NFA), is a 5-tuple $M = (Q,$

q_0, A, δ), where Q, Σ, q_0, and A are defined as for FAs, and the transition functi

is

$$\delta : Q \times \Sigma \rightarrow \wp(Q) \quad (3.$$

As usual, $\wp(Q)$ means the power set of a set Q.

Definition 9 (Nondeterministic Finite Automaton with Λ-transitions) *[1*

A nondeterministic finite automaton with Λ-transitions (abbreviated NFA-Λ) is 5-tuple $M = (Q, \Sigma, q_0, A, \delta)$, where Q, Σ, q_0, and A are defined as for FAs, and tl transition function is

$$\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow \wp(Q) \quad (3.$$

$\wp(Q)$ means the power set of a set Q.

The above three models of finite automata are equivalent. The following theorems te us the connection between finite automata and regular expressions. In the proof Kleene's theorem [10], structural induction is used, that is, construct a finite autom ton equivalent to the given extended regular expression by combining finite automa which is corresponding to the subexpressions of the given extended regular expressio

Theorem 1 (Kleene's Theorem) *[10]*

Any regular language can be accepted by a finite automaton.

The extended regular expression is obtained by adding the intersection and compl mentation operators to the regular expression. Kleene's theorem has showed that a FA accepting the regular expression can be constructed. In order to construct an F equivalent to the given extended regular expression, we only need to show that tl complementation and intersection can be accepted by an FA. As in Kleene's theoren structural induction is used.

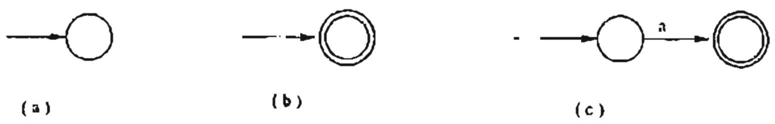

Figure 3.1: NFA-Λs for $\emptyset$, $\{\Lambda\}$, $\{a\}$.

Theorem 2 *Any language denoted by an extended regular expression can be accepte by a finite automaton.*

Proof: It is sufficient to show any language obtained by an extended regular ex pression can be accepted by an NFA-Λ. The set of languages obtained by extende regular expressions over the alphabet Σ is defined to be the smallest set of languag containing the basic languages $\emptyset$, $\{\Lambda\}$, and each of the languages $\{a\}$ $(a \in \Sigma)$. Th class of regular sets is closed under the operations of union, concatenation, *Kleene'* complementation, and intersection. Using structural induction to prove that any la guage denoted by an extended regular expression over an alphabet Σ can be accepte by an NFA-Λ, we must show that the three basic languages can be accepted by a NFA-Λ, and that if L_1 and L_2 are languages that can be accepted by an NFA-Λ, the their union, concatenation, *Kleene**, complementation, and intersection can also b accepted by an NFA-Λ.

NFA-Λs for the three basic languages are obvious. They are shown in Figure 3.1.

Suppose that L_1 and L_2 are accepted by the *NFA*-Λs M_1 and M_2, respectively. M and M_2 are defined as the following:

$M_i = (Q_i, \Sigma, q_i, A_i, \delta_i)$, where $1 \leq i \leq 2$.

Without loss of generality, we may assume that $Q_1 \cap Q_2 = \emptyset$ (by renaming states,

necessary). We will construct NFA-Λs M_u, M_c, M_k, M_{com}, and M_{in}, recognizing th

language $L_1 \cup L_2$, $L_1 L_2$, L_1^*, Σ^* - L_1, $L_1 \cap L_2$, respectively.

The constructions of M_u, M_c, and M_k are given in Kleene's theorem. Here we describ

the constructions of M_{com} and M_{in}.

(1) Construction of $M_{com} = (Q_{com}, \Sigma, q_{com}, A_{com}, \delta_{com})$. Since L_1 is accepted by th

NFA-Λ $M_1 = (Q_1, \Sigma, q_1, A_1, \delta_1)$, by the equivalence of NFA-Λ and DFA, L_1 can b

accepted by a DFA.

Let $M_{d1} = (Q_{d1}, \Sigma, q_{d1}, A_{d1}, \delta_{d1})$, where

- $Q_{d1} = \wp(Q_1)$, $\wp(Q_1)$ means the power set of a set Q_1,
- $q_{d1} = q_1$,
- $A_{d1} = A_1$, and
- $\delta_{d1} = \delta_{d1}$.

be a DFA recognizing L_1, then M_{com} can be obtained by swapping the acceptin

states with the non-accepting states of M_{d1}. That is,

$M_{com} = (Q_{com}, \Sigma, q_{com}, A_{com}, \delta_{com})$, where

- $Q_{com} = Q_{d1}$,
- $q_{com} = q_{d1}$,

- $A_{com} = Q_{d1}\text{-}A_{d1}$, and

- $\delta_{com} = \delta_{d1}$.

(2) Construction of M_{in}. There are two methods to construct M_{in}.

The first one is to construct M_{in} directly. Since L_1 and L_2 are accepted by the NFA-/ M_1 and M_2, respectively, L_1 and L_2 can be recognized by the DFAs.

Let $M_{d1} = (Q_{d1}, \Sigma, q_{d1}, A_{d1}, \delta_{d1})$ and $M_{d2} = (Q_{d2}, \Sigma, q_{d2}, A_{d2}, \delta_{d2})$ be two DF recognizing L_1 and L_2, respectively.

Then $M_{in} = (Q_{in}, \Sigma, q_{in}, A_{in}, \delta_{in})$, where

- $Q_{in} = Q_{d1} \times Q_{d2}$

- $q_{in} = [q_{d1}, q_{d2}]$

- $A_{in} = A_{d1} \times A_{d2}$

- δ_{in} is defined as follows:

$$\delta([p_1, p_2], a) = [\delta_1(p_1, a), \delta_2(p_2, a)], \qquad (3.$$

for all $p_1 \in Q_{d1}$, $p_2 \in Q_{d2}$, and a $\in \Sigma$.

The second method is based on De Morgan's law.

De Morgan's Law [10]:

$$L_1 \cap L_2 = \neg((\neg(L_1)) \cup (\neg(L_2))) \qquad (3.4$$

Since languages are sets, all set operations on languages are inherited from those o sets. De Morgan's law applies to sets, so it also applies to languages. If L_1 and L_2 ar regular languages, then their complements $\neg(L_1)$ and $\neg(L_2)$ are regular language Since the regular sets are closed under union, $(\neg(L_1))\cup(\neg(L_2))$ is regular. Henc $\neg((\neg(L_1))\cup(\neg(L_2)))$ is regular. Therefore, $L_1 \cap L_2$ is regular.

Since L_1 and L_2 are accepted by NFA-Λs M_1 and M_2, respectively, from the construc tion of M_{com}, we can construct FAs M_{com1} and M_{com2} accepting $\neg(L_1)$ and $\neg(L_2$ respectively. By Kleene's Theorem, an NFA-Λ M_{u1} accepting $(\neg(L_1))\cup(\neg(L_2))$ ca be constructed. Using the construction of M_{com} again, we can construct an FA M accepting $\neg((\neg(L_1))\cup(\neg(L_2)))$. Obviously, M' accepts $L_1 \cap L_2$.

3.1.2 Reachability Problem

The two most common computational representations of graphs are adjacency list and adjacency matrices.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array *Adj* $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list *Adj*$[u]$ contair all the vertices v such that there is an edge $(u, v) \in E$. For the adjacency-matri representation of a graph $G = (V, E)$, we assume that the vertices are numbere

1,2,..., $|V|$ in some arbitrary manner. The adjacency-matrix representation of a grap

G that consists of a $|V|\times|V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } \{\mathrm{i, j}\} \in E, \\ 0 & \text{otherwise.} \end{cases} \qquad (3.5$$

For any constructed FA $M = (Q, \Sigma, q, A, \delta)$ that recognizes the given extende regular expressions, if M has k final states, i.e., $k = |A|$, where $k \geq 1$, then we ma convert M into an NFA-Λ M' with exactly one final state F by setting $F = \{\delta(F$ $\Lambda) \mid F_i \in A \}$.

Without loss of generality, we may assume that the first vertex in the graph is th start state of the NFA-Λ, the last vertex in the graph is the final state of the NFA-Λ If there exists a path from the first vertex reachable to the last vertex in the grap i.e., there exists a string accepted by the NFA-Λ, then the language denoted by th extended regular expressions is not empty. Otherwise, the language denoted by th extended regular expressions is empty.

We use the depth-first search algorithm to solve the reachability problem. The deptl first search algorithm is described as follows [13]:

```
DFS($G$)

  1. for each vertex $u \in V[G]$
  2.   do $color[u] \leftarrow$ WHITE
  3.      $\pi[u] \leftarrow$ NIL
```

```
    4. $time \leftarrow 0$
    5. for each vertex $u \in V[G]$
    6.   do if $color[u] =$ WHITE
    7.     then DFS-Visit($u$)

DFS-Visit($u$)
    1. $color[u] \leftarrow$ GRAY
    2. $d[u] \leftarrow time \leftarrow time + 1$
    3. for each $v \in Adj[u]$
    4.   do if $color[v] =$ WHITE
    5.     then $\pi[v] \leftarrow u$
    6.       DFS-Visit($v$)
    7. $color[u] \leftarrow$ BLACK
    8. $f[u] \leftarrow time \leftarrow time + 1$
```

In the depth-first search, each vertex is initially white, becomes grey when it is dis
covered in the search, and becomes black when it is finishes. Let the start state of th
NFA be the root of a new tree in the depth-first forest. If the final state of the NF
is discovered or finished, i.e., the color of the final state of the NFA is changed fron
white to grey or black, after the depth-first search algorithm is done, then there exis

a path from the start state to the final state of the NFA, i.e., there exists a strin
accepted by the NFA, and the language denoted by the extended regular expression
is not empty, Otherwise, the language denoted by the extended regular expression
is empty.

3.2 A Complexity Analysis

Let us consider the time and space complexities of our algorithm.

In the procedure of constructions of FAs, since in Kleene's theorem, NFA is use
for the union, concatenation and Kleene closure operators, the number of states i
additive. The complementation operator requires a conversion from an NFA to
DFA, so the number of states increases exponentially. For the intersection operato
if we construct the FA using the first method, then the number of states increases a
the product. If we use the second method which depends on the complementatio
operator, then the number of states increases exponentially. So the procedure c
constructions of FAs for extended regular expressions requires at least exponentia
time and space.

Considering the time complexity of the solution for the reachability problem, let |V
be the number of vertices, and |E| be the number of edges in the graph $G(V, E$
then the running time of depth-first search algorithm is $\Theta(|V| + |E|)$. For the spac
complexity of the solution for the reachability problem, obviously, the adjacency lis
requires $O(|V| + |E|)$.

Our algorithm to solve the *Emptiness Problem for Extended Regular Expression* consists of (1)constructions of FAs and (2) the reachability algorithm. Since con structions of FAs require exponential time and space, our algorithm for solving th *Emptiness Problem for Extended Regular Expressions* requires at least exponentia time and space.

3.3 Conclusions

In this thesis, we solve the *Emptiness Problem for Extended Regular Expressions* b constructing FAs equivalent to the given extended regular expressions and applyin depth-first search algorithm on the reachability problem.

The procedure of constructions of FAs works by creating NFA-Λ for the three basi languages ∅, $\{\Lambda\}$, and $\{a\}$ and then constructing FAs for the union, concatenation and intersection of two languages accepted by FAs, and the Kleene closure, and com plementation of a language accepted by FAs. By structural induction, all language denoted by extended regular expressions over the alphabet Σ can be accepted by a FA.

For the procedure of solution to the reachability problem, all vertices can be reache from the start state of the NFA will be discovered in the depth-first search algorithm The algorithm will explore all the possible paths that exist between the start stat and the final state of the NFA. If there is a path from the start state to the fina state, then the algorithm can find it.

By analyzing the time and space complexity of our algorithm, we conclude that th *Emptiness Problem for Extended Regular Expressions* is intractable and it requires a least exponential time and space.

Bibliography

[1] AHO, HOPCROFT, AND ULLMAN. NP-complete problems. *The Design an Analysis of Computer Algorithm* (1974), pp. 395-423.

[2] G. L. MILLER. Riemann's hypothesis and tests for primary. *J. Comput. Systen Sci.* (1976), pp. 300-317.

[3] H. B. HUNT. On the time and tape complexity of languages. *Doctoral Thesis Dept. of Computer Science, Cornell University, Ithaca, NY.* (1973).

[4] H. B. HUNT, AND T. G. SZYMANSKI. Complexity metatheorems for context free grammar problems. *J. Comput. System Sci. 13* (1976), pp. 318-334.

[5] J. S. VITTER, AND R. A. SIMONS. Parallel algorithms for unification anc other complete problems in P. In *Proceedings of the 1984 Annual Conference o the ACM on the Fifth Generation Challenge* (1984), pp. 75-84.

[6] L. J. STOCKMEYER. Planar 3-colorability is NP-complete. *SIGACT New* (1973), pp. 19-25.

[7] M. R. GAREY, D. S. JOHNSON, AND L. STOCKMEYER. Some simplified NP-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing* (1974), pp. 47-63.

[8] M. R. GAREY, AND D. S. JOHNSON. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[9] M. AGRAWAL, N. KAYAL, AND N. SAXENA. Primes is in P. *http://www.cse.iitk.ac.in/news/primality.html* (2002).

[10] M. SIPSER. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[11] S. HOMER, AND A. L. SELMAN. *Computability and Complexity Theory.* Springer, 2001.

[12] S. R. BUSS. The polynomial hierarchy and fragments of bounded arithmetic. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* (1985), pp. 285-290.

[13] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST. *Introduction to Algorithms.* The MIT Press, 1999.

[14] V. PRATT. Every prime has a succinct certificate. *SIAM J. Comput.* (1975), pp. 214-220.

[15] W. J. SAVITCH. Relationship between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.* (1970), pp. 177-192.

[16] W. J. SAVITCH. Nondeterministic log n space. In *Proceedings 8th Ann. Prince- ton Conf. on Information Science and Systems* (1974), pp. 21-23.

Appendix A

A Program to Solve the Emptiness Problem for Extended Regular Expressions

A PROGRAM TO SOLVE THE EMPTINESS PROBLEM FOR EXTENDED REGULAR EXPRESSIONS

The *emptiness problem for extended regular expressions* is described as follows:

Instance: Given a extended regular expression R over the alphabet Σ.

Question: Is the language denoted by R empty.

The algorithm consists of two parts:

1. constructions of finite automata for the given extended regular expressions
2. the depth_first search algorithm to solve the reachability problem

The algorithm is implemented in a C/C++ program. The program includes two files: pgm1.h and main.cpp.

PROGRAMMERS' GUIDE

This program is to solve the emptiness problem for extended regular expressions. It is implemented in Microsoft Visual C++. The program consists of two files: one head file (pgm1.h) and one C/C++ source code file (main.cpp). In the program, we define one class, two structures, one main function and twenty-seven subroutines.

There are two following structures:

1. State: it holds four elements: start, final, value and color. Start is a character to check the state is the start state of the finite automaton or not. If the state is the start state, then we assign 'Y' to the element state, else we assign 'N' to the element state. Final is a character to check the state is the final state of the finite automaton or not. If the state is the final state, then we assign 'Y' to the element final, else we assign 'N' to the element final. Value is a character to give the input data of states. In the program, let us assume that the symbol '$' is not in the alphabet of extended regular expressions, then we initialize values of all states in the finite automaton to be '$'. The above three elements are used in the constructions of FAs. The fourth element of the state structure is color, which is a character to assign the color to every state in the finite automaton. It could be 'w', 'g', or 'b': 'w' means white; 'g' means grey; 'b' means black. The element color is used in the depth_first search algorithm to solve the reachability problem.

PROGRAMMERS' GUIDE

This program is to solve the emptiness problem for extended regular expressions. It is implemented in Microsoft Visual C++. The program consists of two files: one head file (pgm1.h) and one C/C++ source code file (main.cpp). In the program, we define one class, two structures, one main function and twenty-seven subroutines.

There are two following structures:

1. State: it holds four elements: start, final, value and color. Start is a character to check the state is the start state of the finite automaton or not. If the state is the start state, then we assign 'Y' to the element state, else we assign 'N' to the element state. Final is a character to check the state is the final state of the finite automaton or not. If the state is the final state, then we assign 'Y' to the element final, else we assign 'N' to the element final. Value is a character to give the input data of states. In the program, let us assume that the symbol '$' is not in the alphabet of extended regular expressions, then we initialize values of all states in the finite automaton to be '$'. The above three elements are used in the constructions of FAs. The fourth element of the state structure is color, which is a character to assign the color to every state in the finite automaton. It could be 'w', 'g', or 'b': 'w' means white; 'g' means grey; 'b' means black. The element color is used in the depth_first search algorithm to solve the reachability problem.

2. Fa: it holds two elements: node and size. Node is a two-dimension array. Every element in the array is a structure of state. Size is an integer denoting the number of states in a finite automaton.

There is one class named FA: it holds a variable and twenty-one functions.

1. The variable called m_fa which is a pointer pointing to a structure of fa.
2. Twenty-one functions are described as follows:
   - FA( ) – it creates the constructor function
   - ~FA( ) – it creates the destructor function
   - GetUserInput (char *, char *) – it gets the alphabet and extended regular expressions from the input
   - Get_Array(int *, char *) – it decides the highest priority of operators in the extended regular expression
   - Get_Max(int *, char *) – it gets the maximum in an integer array
   - Get_Left(char *, char *, int) – it gets the left hand side of the k elements in a given sentence of size 3
   - Get_Right(char *, char *, int) – it gets the right hand side of the k elements in a given sentence of size 3
   - Get_Left_Op(char *, int *, char *, int) – it gets the left hand side of the k elements in a given sentence
   - Get_Right_Op(char *, int *, char *, int) – it gets the right hand side of the k elements in a given sentence
   - Is_In_Sigma (char *, char) -- it checks a character is in the alphabet or not

- Transfer_Basic (fa *, char *, char) – it transfers the three basic languages into a structure of fa
- Get_C_Value (int n, int m) – it calculates the choice number which is the number of ways of picking m unordered outcomes from n possibilities
- Get_C_Sum_M (int n, int m_b, int m_e) – it calculates the sum of some choice numbers, n is unchanged
- Get_C_Sum_N (int n_b, int n_e, int m) – it calculates the sum of some choice numbers, m is unchanged
- Get_Index (int *, int, int) – it calculates the index in a DFA corresponding to an NFA
- Sort_Array (int *, int) – it sorts one array in increasing order
- NFA_To_DFA (fa *, fa *, char *) – it transfers an NFA to a DFA by using the subset construction
- Union (fa *, fa *, fa *) – it does the union operation on two structures of fa
- Concatenation (fa *, fa *, fa *) – it does the concatenation operation on two structures of fa
- Star (fa *, fa *) – it does the Kleene star operation on one structure of fa
- Complementation (fa *, fa *) – it does the complementation operation on one structure of fa

- Intersection (fa *, fa *, fa *) – it does the intersection operation on two structures of fa
- Func1 (char *, char *, int) – it does the operation on the highest priority operator from an input string
- Func2 (fa *, char , fa *, fa *, char *) – it does five operators for the given extended regular expressions
- Func0 (fa *, char *, char *, int) – it transfers the extended regular expressions to a "tree" structure and calls the corresponding subroutines to do five operations on structures of fa
- DFS Visit (fa *, int) – it changes the color of vertices on the path starting from the start state of the finite automaton, it is used for function DFS
- DFS (fa *) – it uses depth-first search to solve the reachbility problem

There is one main function: it inputs the alphabet and extended regular expressions and output the given extended regular expression is empty or not.

The program requires exponential time and space.

USERS' MANUAL

This program is using C/C++ to solve the emptiness problem for extended regular expressions. The operating system is Windows XP/2000/98/95. It runs in Microsoft Visual C++ (Version 6.0).

Users may use the tools of the menu bar in Microsoft Visual C++ (Version 6.0) to compile and execute the program. After clicking the "!" button in the tool bar, the program executes, and one window to get the input and show the output comes out on the screen.

In the program, we use the following denotation when defining the five operations on extended regular expressions:

- 'U' denotes union
- 'I' denotes intersection
- 'C' denotes concatenation
- '!' denotes implement
- '*' denotes star

The input of the program consists of two parts: the alphabet and extended regular expressions. Such as the follows:

1. In the program, the alphabet could be any character except '$' and five operators that are 'U', 'I', 'C', '!', and '*'. For example: "abcdefg", "1234567", etc.
   - One message "please enter the alphabet" to ask users to input the alphabet.
   - Users may enter (1) "abc", then press the "return" key.

or (2) "a" then press the "return" key.

2. In the program, we suppose that every extended regular expression is fully parenthesized. For example: ((aUb)I(aUc)), (a*), (b!), etc.

   - One message "please enter the extended regular expression" to ask users to input the extended regular expression.
   - Users may enter (1) "((aUb)I(aUc))" (or "a!") then press the "return" key.

     or (2) "a!" then press the "return" key.

The output of the program is one message to show that the extended regular expression is empty or not. Such as the output of the above input will be:

- (1) The extended regular expression is not empty.
- (2) The extended regular expression is empty.

```
//=================================================================
//pgm1.h
//pgm1.h is the head file of the program.
//It defines all classes, structures, and functions.
//In this program, there is one class: FA,
//two structures: state and fa,
//one main function, and twenty-one called functions.
//=================================================================

//=================================================================
//Define a structure named state which holds four elements:
//The first three elements are: start, final, and value
//start is a character to check the state is the start
//state of the finite automaton or not:
//if the state is the start state, then we assign 'Y' to
//the element state, else we assign 'N' to the element state.
//final is a character to check the state is the final
//state of the finite automaton or not:
//if the state is the final state, then we assign 'Y' to
//the element final, else we assign 'N' to the element final.
//value is a character to give the input value of states:
//Let us assume that the symbol '$' is not in the alphabet
//of extended regular expressions. Then we initialize the value
//of all states in the finite automaton '$'.
//The above three elements are used in the constructions of FAs.
//The last element of the state structure is color,
//which is a character to assign the color to every state in
//the finite automaton. It could be 'w', 'g', or 'b'.
//'w' means white; 'g' means grey; 'b' means black.
//color is used in the depth_first search algorithm to
//solve the reachability problem
//=================================================================

typedef struct state{
        char start;
        char final;
        char value;
        char color;

}state;

//=================================================================
//Define a structure named fa which holds two elements:
//node and size. Node is a two-dimension array. Every element
//in the array is a structure of state.
//Size is an integer denoting the number of states in a fa
```

```
//==========================================================

typedef struct fa{
        state node[80][80];
        int size;
}fa;

//==========================================================
//Define a class named FA.
//This class has a public variable m_fa
//and twenty-one functions
//==========================================================

class FA {
public:
        fa* m_fa;

public:

//==========================================================
//constructor and destructor
//==========================================================

        FA();
        ~FA();

//==========================================================
//five operator functions: Union, Concatenation,
//Star, Complementation, and Intersection
//==========================================================

        void Union (fa* pM3, fa* pM1, fa* pM2);
        void Concatenation(fa* pM3, fa* pM1, fa* pM2);
        void Star(fa* pM3, fa* pM1);
        void Complementation(fa* pM2, fa* pM1);
        void Intersection(fa* pM3, fa* pM1, fa* pM2);

//==========================================================
//function NFA_To_DFA is to convert an NFA to a DFA
//==========================================================

        void NFA_To_DFA (fa* pMd, fa* pMn, char* Sigma);

//==========================================================
//functions Func0, Func1 and Func2 are used to
//transfer the extended regular expressions
```

```
//===================================================================

typedef struct fa{
        state node[80][80];
        int size;
}fa;

//===================================================================
//Define a class named FA.
//This class has a public variable m_fa
//and twenty-one functions
//===================================================================

class FA {
public:
        fa* m_fa;

public:

//===================================================================
//constructor and destructor
//===================================================================

        FA();
        ~FA();

//===================================================================
//five operator functions: Union, Concatenation,
//Star, Complementation, and Intersection
//===================================================================

        void Union (fa* pM3, fa* pM1, fa* pM2);
        void Concatenation(fa* pM3, fa* pM1, fa* pM2);
        void Star(fa* pM3, fa* pM1);
        void Complementation(fa* pM2, fa* pM1);
        void Intersection(fa* pM3, fa* pM1, fa* pM2);

//===================================================================
//function NFA_To_DFA is to convert an NFA to a DFA
//===================================================================

        void NFA_To_DFA (fa* pMd, fa* pMn, char* Sigma);

//===================================================================
//functions Func0, Func1 and Func2 are used to
//transfer the extended regular expressions
```

```
//to  a "tree" structure and call the corresponding
//subroutines to do the operation on five operators
//===========================================================

        void Func0 (fa* result0, char* Sigma0, char* sentence0, int length0);
        struct fa* Func1 (char* Sigma1, char* sentence, int length) ;
        void Func2(fa* pRes,char op, fa* pL, fa* pR, char* Sigma);

//===========================================================
//function GetUserInput is to get the alphabet
//and extended regular expressions
//===========================================================

        void GetUserInput(char* sigma, char* input);

//===========================================================
//function Get_Array, Get_Max, Get_Left, Get_right,
//Get_Left_Op, and Get_Right_Op are to get the left_hand
//side and the right_hand side of the operator in
//a given sentence
//===========================================================

        void Get_Array(int* array1, char* sentence1);
        int Get_Max(int* array2, char* sentence2);
        void Get_Left(char* LHS, char* sentence2, int k);
        void Get_Right(char* RHS, char* sentence3, int k);
        void Get_Left_Op(char* LHS, int* array, char* sentence2, int k);
        void Get_Right_Op(char* RHS, int* array, char* sentence2, int k);

//===========================================================
//functions Get_C_Value, Get_C_Sum_M, Get_C_Sum_N, and Get_Index
//are to get the index in the DFA corresponding to the NFA
//===========================================================

        int Get_C_Value(int n, int m);
        int Get_C_Sum_M(int n, int m_b, int m_e);
        int Get_C_Sum_N(int n_b, int b_e, int m);
        int Get_Index(int *p, int size, int n);

//===========================================================
//function Transfer_Basic is to transfer the three basic
//language into fa structures
//===========================================================

        void Transfer_Basic(fa* pM, char *Sigma, char c);
```

```
//=====================================================================
//function Is_In_Sigma is to check a character is
//in the alphabet or not
//=====================================================================

        bool Is_In_Sigma(char *Sigma, char c);

//=====================================================================
//function Sort_Array is to sort an array in
//increasing order
//=====================================================================

        void Sort_Array(int*p, int size);

//=====================================================================
//functions DFS_Visit and DFS use depth-first search
//to solve the reachability problem
//=====================================================================

        void DFS_Visit(fa* pMM, int u);
        void DFS(fa* pMM);

};
```

```
//================================================================
//main.cpp
//This program is implemented in Microsoft Visual C++ 6.0
//After compiling and executing the program, one
//message "please enter the alphabet" appears
//on the screen. The user enters the alphabet
//such as "abcdefg" and then presses the enter key.
//Another message "please enter the extended regular
//expressions" comes out. The user can enter the
//extended regular expressions and presses the enter
//key. Finally, after the user enter
//the alphabet and extended regular expressions,
//the program will output whether the languages denoted by the
//extended regular expressions
//is empty or not.
//================================================================

#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "pgm1.h"

//================================================================
// Functions declaration
//================================================================

void Union (fa* pM3, fa* pM1, fa* pM2);
void Concatenation(fa* pM3, fa* pM1, fa* pM2);
void Star(fa* pM3, fa* pM1);
void Complementation(fa* pM2, fa* pM1);
void Intersection(fa* pM3, fa* pM1, fa* pM2);

void NFA_To_DFA (fa* pMd, fa* pMn, char* Sigma);

void Func0 (fa* result0, char* Sigma0, char* sentence0, int length0);
struct fa* Func1 (char* Sigma1, char* sentence, int length) ;
void Func2(fa* pRes,char op, fa* pL, fa* pR, char* Sigma);

void GetUserInput(char* sigma, char* input);
void Get_Array(int* array1, char* sentence1);
int Get_Max(int* array2, char* sentence2);
void Get_Left(char* LHS, char* sentence2, int k);
void Get_Right(char* RHS, char* sentence3, int k);
void Get_Left_Op(char* LHS, int* array, char* sentence2, int k);
void Get_Right_Op(char* RHS, int* array, char* sentence2, int k);

int Get_C_Value(int n, int m);
int Get_C_Sum_M(int n, int m_b, int m_e);
int Get_C_Sum_N(int n_b, int b_e, int m);
int Get_Index(int *p, int size, int n);

void Transfer_Basic(fa* pM, char *Sigma, char c);
bool Is_In_Sigma(char *Sigma, char c);
void Sort_Array(int*p, int size);

void DFS_Visit(fa* pMM, int u);
void DFS(fa* pMM);

//===========================================================
//constructor function
//===========================================================

FA::FA() {

}

//===========================================================
//destructor function
//===========================================================

FA::~FA() { }

//===========================================================
//main function: its input is sigma and extended regular expressions,
//its output is the given extended regular expression is empty or not
//===========================================================

int main(void)
{
        char sigma[80] = {0};
        char input[80] = {0};

        GetUserInput(sigma, input);

        fa* rfa = new fa;
        FA* pFA = new FA();

        if(strlen(input) == 1)  //for the case of input which is only one character
        {
            if((Is_In_Sigma(sigma, input[0]) == true) && (input[0] != 'O'))
    //O means empty set
                    cout << "the extended regular expression is not empty" << '\n';
            else
                    cout << "the extended regular expression is empty" << '\n';
    }
    else
    {
    for(int i=0; i<80; i++)
            for(int j=0; j<80; j++)
            {
                    rfa->node[i][j].final = 'N';
                    rfa->node[i][j].start = 'N';
                    rfa->node[i][j].value = '$';
    // $ denote initial value and it is not in the sigma

            }

    Func0(rfa, sigma, input, strlen(sigma));
    DFS(rfa);

    int flag = 0;
    for(int rr = 0; rr < rfa->size; rr++)
    {
            if((rfa->node[rr][rr].color != 'w') && (rfa->node[rr][rr].final == 'Y'))
                    flag = 1;
    }
    if (flag == 1)
            cout << "the extended regular expression is not empty" << '\n';
    else
            cout << "the extended regular expression is empty" << '\n';

  }

    pFA->~FA();
    delete pFA;
    pFA = 0;
    return 0;

}

//=====================================================================
//Function GetUserInput is to get the alphabet and extended regular
//expressions
//=====================================================================
            if((Is_In_Sigma(sigma, input[0]) == true) && (input[0] != 'O'))
        //O means empty set
                    cout << "the extended regular expression is not empty" << '\n';
            else
                    cout << "the extended regular expression is empty" << '\n';

        }
        else
        {
        for(int i=0; i<80; i++)
            for(int j=0; j<80; j++)
            {
                    rfa->node[i][j].final = 'N';
                    rfa->node[i][j].start = 'N';
                    rfa->node[i][j].value = '$';
        // $ denote initial value and it is not in the sigma

            }

        Func0(rfa, sigma, input, strlen(sigma));
        DFS(rfa);

        int flag = 0;
        for(int rr = 0; rr < rfa->size; rr++)
        {
            if((rfa->node[rr][rr].color != 'w') && (rfa->node[rr][rr].final == 'Y'))
                    flag = 1;
        }
        if (flag == 1)
            cout << "the extended regular expression is not empty" << '\n';
        else
            cout << "the extended regular expression is empty" << '\n';

    }

        pFA->~FA();
        delete pFA;
        pFA = 0;
        return 0;

}

//=====================================================================
//Function GetUserInput is to get the alphabet and extended regular
//expressions
//=====================================================================
void GetUserInput(char* sigma, char* input)
{
        char buff[80] = {0};

        cout << "please enter an alphabet" << endl;
        cin.getline(buff,80,'\n');
        strcpy(sigma, buff);
        cout << "please enter an extended regular expression" << endl;
        cin.getline(buff, 80, '\n');
        strcpy(input, buff);

}

//=====================================================================
// Function Get_Array is to decide the highest priority of operators
// in the extended regular expression
//=====================================================================

void Get_Array(int* array1, char* sentence1)
{
        int len = strlen(sentence1);
        for (int index =0; index < len; index++)
        {
                if (sentence1[index] == '(')
                array1[index] = array1[index-1] + 1;
                else if (sentence1[index] == ')')
                        array1[index] = array1[index-1] - 1;
                else
                        array1[index] = array1[index-1];
        }
}

//=====================================================================
// Function Get_Max is to get the maximum in an integer array
//=====================================================================

int Get_Max(int* array2, char* sentence2)
{
        int len = strlen(sentence2);
        int max_value = -1;
        for(int i=0; i<len; i++)
        {
                if(array2[i] > max_value)
                        max_value = array2[i];
        }
        return max_value;
```

```
}

//===================================================================
// Function Get_Left is to get the left hand side of the k elements
// in a given sentence of size 3
//===================================================================

void Get_Left(char* LHS, char* sentence2, int k)
{
        char left[80] = {0};
        for (int m=0; m<=k-2; m++)
        left[m] = sentence2[m+1];
         for (int mm = k-1; mm<80; mm++)
        left[mm] = '\0';

        strcpy(LHS, left);
}

//===================================================================
// Function Get_Right is to get the right hand side of the k elements
// in a given sentence of size 3
//===================================================================

void Get_Right(char* RHS, char* sentence3, int k)
{
        int len = (int)strlen(sentence3);
        char right[80] = {0};
        for(int n=k+1; n<len-1; n++)
        right[n-k-1] = sentence3[n];
        for(int nn = len-k-2; nn <80; nn++)
        right[nn] = '\0';

        strcpy(RHS, right);
}

//===================================================================
// Function Get_Left_Op is to get the left hand side of the k elements
// in a given sentence
//===================================================================

void Get_Left_Op(char* LHS, int* array, char* sentence2, int k)
{
        char left[80] = {0};
        int i=k;
        int index = 0;
        while(i>0)
        {
                if(array[i] >= array[k])
                        i--;
                else
                        exit(0);
        }
        index = i;
        for(int j= index+1; j<k; j++)
                left[j-index-1] = sentence2[j];
        for(int jj = k-index-1; jj<80; jj++)
                left[jj] = '\0';

        strcpy(LHS, left);
}

//===========================================================================
// Function Get_Right_Op is to get the right hand side of the k elements
// in a given sentence
//===========================================================================

void Get_Right_Op(char* RHS, int* array, char* sentence2, int k)
{
        char right[80] = {0};
        int i=k;
        int index = 0;
        while(i< (int)strlen(sentence2))
        {
                if(array[i] >= array[k])
                        i++;
                else
                        break;
        }
                index = i;
        for(int j= k+1; j<index; j++)
                right[j-k-1] = sentence2[j];
        for(int jj = index -k; jj<80; jj++)
                right[jj] = '\0';

        strcpy(RHS, right);
}

//===========================================================================
// Function Is_In_Sigma is to check a character is in the alphabet or not
//===========================================================================
bool Is_In_Sigma(char *Sigma, char c)
{
        int len = strlen(Sigma);
        for (int i=0; i<len; i++)
                if((Sigma[i] == c )|| (c == 'e'))
                        return true;
        return false;
}

//==========================================================================
// Function Transfer_Basic is to transfer the three basic languages
// into an fa structure
//==========================================================================

void Transfer_Basic(fa* pM, char *Sigma, char c)
{
        for (int i=0; i<2; i++)
         for (int j=0; j<2; j++)
                {
                 pM->node[i][j].start = 'N';
                 pM->node[i][j].value = '$';
                 pM->node[i][j].final = 'N';
                }

        if(c == 'e')  { //empty string which is epsilon
                pM->node[0][0].start = 'Y';
                pM->node[0][1].value = 'e';
                pM->node[1][1].final = 'Y';
        }
        else if(c == 'O')   //"O" denotes empty set
                pM->node[0][0].start = 'Y';
        else if(Is_In_Sigma(Sigma, c)) {
                pM->node[0][0].start = 'Y';
                pM->node[0][1].value = c;
                pM->node[1][1].final = 'Y';
        }
        pM->size = 2;
}

//==========================================================================
// Function Union is to do the union operator on two fa structures
//==========================================================================

void Union (fa* pM3, fa* pM1, fa* pM2)
{
        int rr=0, cc=0;
        int r1 = pM1->size;
        int r2 = pM2->size;

        for (rr = 1; rr < r1+1; rr++)
                for (cc = 1; cc < r1+1; cc++)
                        pM3->node[rr][cc].value = pM1->node[rr-1][cc-1].value;
        pM3->node[0][1].value = 'e';
        pM3->node[0][r1+1].value = 'e';
        for(rr = r1+1; rr < r1+r2+1; rr++)
                for(cc = r1+1; cc < r1+r2+1; cc++)
                        pM3->node[rr][cc].value = pM2->node[rr-r1-1][cc-r1-1].value;

        pM3->node[r1][r1+r2+1].value = 'e';
        pM3->node[r1+r2][r1+r2+1].value = 'e';
        pM3->node[0][0].start = 'Y';
        pM3->node[r1+r2+1][r1+r2+1].final = 'Y';
        pM3->node[1][1].start = 'N';
        pM3->node[r1+1][r1+1].start = 'N';
        pM3->node[r1][r1].final = 'N';
        pM3->node[r1+r2][r1+r2].final = 'N';
        pM3->size = pM1->size + pM2->size + 2;
}

//=========================================================================
// Function Concatenation is to do the concatenation operator on
// two fa structures
//=========================================================================

void Concatenation(fa* pM3, fa* pM1, fa* pM2)
{
        int rr, cc;
        int r1 = pM1->size;
        int r2 = pM2->size;

        for (rr = 1; rr < r1+1; rr++)
                for (cc = 1; cc < r1+1; cc++)
                        pM3->node[rr][cc].value = pM1->node[rr-1][cc-1].value;
        pM3->node[r1][r1+1].value = 'e';
        for(rr = r1+1; rr < r1+r2+1; rr++)
                for(cc = r1+1; cc < r1+r2+1; cc++)
                        pM3->node[rr][cc].value = pM2->node[rr-r1-1][cc-r1-1].value;

        pM3->node[r1+r2][r1+r2+1].value = 'e';
        pM3->node[0][1].value = 'e';

        pM3->node[0][0].start = 'Y';
        pM3->node[r1+r2+1][r1+r2+1].final = 'Y';

        pM3->node[1][1].start = 'N';
        pM3->node[r1+1][r1+1].start = 'N';
        pM3->node[r1][r1].final = 'N';
        pM3->node[r1+r2][r1+r2].final = 'N';

        pM3->size = pM1->size + pM2->size + 2;

}

//==========================================================
// Function Star is to do the Kleene star operator on one fa structure
//==========================================================

void Star(fa* pM3, fa* pM1)
{
        int rr, cc;
        int r1 = pM1->size;

        pM3->node[0][1].value = 'e';

        pM3->node[r1][1].value = 'e';
        for(rr = 1; rr < r1+1; rr++)
                for(cc = 1; cc < r1+1; cc++)
                        pM3->node[rr][cc].value = pM1->node[rr-1][cc-1].value;
        pM3->node[r1][r1+1].value = 'e';
        pM3->node[0][r1+1].value = 'e';
        pM3->node[0][0].start = 'Y';
        pM3->node[1][1].start = 'N';

        pM3->node[r1+1][r1+1].final = 'Y';
        pM3->node[r1][r1].final = 'N';
        pM3->size = pM1->size + 2;

}

//==========================================================
// Function Complementation is to do the complementation opertor on
// one fa structure
//==========================================================

void Complementation(fa* pM3, fa* pM1)
{
        struct fa* pM2 = new fa;
        int r1 = pM1->size;
```

```
	for(int rr = 0; rr < r1; rr++)
		for(int cc = 0; cc < r1; cc++) {
			if(pM1->node[rr][cc].final == 'Y')
				pM2->node[rr][cc].final = 'N';
			else if(pM1->node[rr][cc].final == 'N')
			 pM2->node[rr][cc].final = 'Y';
			pM2->node[rr][cc].start = pM1->node[rr][cc].start;
			pM2->node[rr][cc].value = pM1->node[rr][cc].value;
		}

	for(int r = 0; r < r1; r++)
		for(int cc = 0; cc < r1; cc++)
		{
			pM3->node[rr+1][cc+1].value = pM2->node[rr][cc].value;
			pM3->node[rr+1][cc+1].final = pM2->node[rr][cc].final;
			pM3->node[rr+1][cc+1].start = 'N';
			if(pM2->node[rr][cc].start == 'Y')
				pM3->node[0][rr+1].value = 'e';
			if(pM2->node[rr][cc].final == 'Y')
				pM3->node[rr+1][pM1->size+1].value = 'e';

		}
	pM3->node[0][0].start = 'Y';
	pM3->node[pM1->size+1][pM1->size+1].final = 'Y';

	pM3->size = pM1->size + 2;

	delete pM2;
}

//=====================================================================
// Function Intersection is to do the intersection operation on two fa
// structures by applying De Morgan rule
//=====================================================================

void Intersection(fa* pM3, fa* pM1, fa* pM2)
{
	struct fa *M3 = new fa;
	struct fa *CM1 = new fa;
	struct fa *CM2 = new fa;

	Complementation(CM1,pM1);
	Complementation(CM2,pM2);
	Union(M3, CM1, CM2);
	Complementation(pM3, M3 );
```

```
}

//------------------------------------------------------------------------
// Function Sort_Array is to sort one array in increasing order
//------------------------------------------------------------------------

void Sort_Array(int* p, int size)
{
        int t=0;
        for(int i 0; i<size - 1; i++)
        {
                if(p[i] > p[i+1])
                {
                  t = p[i];
                  p[i] = p[i+1];
                  p[i+1] = t;
                }
        }
}

//------------------------------------------------------------------------
// Function Get_C_Value is to get choice number which is the number of
// ways of picking m unordered outcomes from n possibilities.
//------------------------------------------------------------------------

int Get_C_Value(int n, int m)
{
        int Num=1;
        int Den=1;
        int Value = 1;

        if((n==0) || (m==0))
                return Value;
        else{
        for(int ii=n; ii > n-m; ii--)
                Num = Num*ii;
        for(int j=m; j>0; j--)
                Den = Den*j;
        Value = (int)(Num/Den);
        return Value;
        }
}

//------------------------------------------------------------------------
// Function Get_C_Sum_N is to get the sum of choice numbers
// from "n_begin choose m" to "n_end choose m"
```

```
//==========================================================================

int Get_C_Sum_N(int n_begin, int n_end, int m)
{
        int sum=0;
        for(int i=n_begin; i> n_end-1; i--)
        sum    sum + Get_C_Value(i, m);
        return sum;
}

//==========================================================================
// Function Get_C_Sum_M is to get the sum of choice numbers
// from "n choose m_begin" to "n choose m_end"
//==========================================================================

int Get_C_Sum_M(int n, int m_begin, int m_end)
{
        int sum=0;
        for(int i=m  begin; i>m_end-1; i--)
                sum = sum + Get_C_Value(n, i);
        return sum;
}

//==========================================================================
// Function Get_Index is to get the index in dfa corresponding to
// the nfa
//==========================================================================

int Get_Index(int* p, int size, int n)
{
 int index =0;
 int s1=0;
 int s2=0;
 int s3=0;

 if(size  = 0)
         index = 0;
 else if(size == 1)
         index = p[0]+1;
 else{
        s1 = Get_C  Sum_M(n, 1, size-1);
        s3 = p[size-1] - p[size-2]; //to get s2
        for(int k=0; k<= size-2; k++)
                        s2 = s2 + Get_C_Sum_N(n-k-1, n-p[k]+1, size-2-k);
         index = s1+s2+s3;
 }
 return index;
}

//=====================================================================
// Funciton NFA_To_DFA is to transfer an NFA to a DFA by using the
// subset construction. It is used in doing the complementation operation
//=====================================================================

void NFA_To_DFA (fa* pMd, fa* pMn, char *Sigma)  {
        int Mn_num = pMn->size;
        int Md_num = (int)pow(2,Mn_num);
        int rr, cc, jj;
        int k=0;

        if(Md_num >= 65536)
        {
                cout << "out of space!" << '\n';
                exit(0);
        }
        for(rr = 0; rr < Md_num; rr++)
                for(cc = 0; cc < Md_num; cc++)
                {
                        pMd->node[rr][cc].start = 'N';
                        pMd->node[rr][cc].final = 'N';
                        pMd->node[rr][cc].value = '$';
                }

         int count = 0;
        int f_count = 0;
        int v_count = 0;
        int u_count = 0;
        int p[20] = {0};
        int f[20] = {0};//final
        int v[100] = {0}; //value
        int u[20] = {0};
        int x[100] = {0};

        int w[20][20] = {0};
        int w_size[20] = {0};
        int r_count = 0;
        int c_count = 0;
        int x_count = 0;

        pMd->size = Md_num;

        for(rr = 0; rr < Mn_num; rr++)
```

```
{
                if(pMn->node[rr][rr].start == 'Y')
                {
                        p[count++] = rr;
                                for(jj = 0; jj < Mn_num; jj++)
                                {
                                        if(pMn->node[rr][jj].value == 'e')
                                        {
                                                if(jj != p[count-1])
                                                {
                                                p[count] = jj;
                                                count++;
                                                }
                                                rr = jj;
                                                jj  0;
                                        } //if
                                } //for jj
                        Sort_Array(p, count);
                        int index_dfa= Get_Index(p, count, Mn_num);
                        pMd->node[index_dfa][index_dfa].start = 'Y';
                } //if
} //for rr

for(rr = 0; rr < Mn_num; rr++)
{
        if(pMn->node[rr][rr].final == 'Y')
        {
                f[f_count++] = rr;
                int index_f_dfa = Get_Index(f, f_count, Mn_num);
                pMd->node[index_f_dfa][index_f dfa].final = 'Y';

                for(k=0; k< Mn_num-1; k++)
                {
                 if(k !=rr)
                        {
                        f[f_count] = k;
                        f_count++;
                        Sort_Array(f, f_count);
                        int index_f_dfa = Get_Index(f, f_count, Mn_num);
                        pMd->node[index_f_dfa][index_f_dfa].final = 'Y';
                        }
                }
        } // if
} // for rr

int S[20] = {0};
int Sub  set[20] = {0};

int index = 0;
int j = 0;
int len = strlen(Sigma);

for(k = 0; k < len; k++)
{
          int flag = 0;
          {
          for(rr = 0; rr < Mn_num; rr++)
          {
                    for(cc = 0; cc < Mn_num; cc++)
                    {
                              if(pMn->node[rr][cc].value == Sigma[k])
                              {
                              flag = 1;
                              v[v_count++] = cc;
                              u[u_count++] = rr;

                              for(jj = 0; jj < Mn_num; jj++)
                              {
                              if(pMn->node[cc][jj].value == 'e')
                              {
                              if(jj != v[v_count-1])
                              {
                              v[v_count] = jj;
                              v_count++;

                              }

                              cc = jj;
                              jj = 0;
                              } //if
                              }//for jj
                              Sort_Array(v, v_count);
                              Sort_Array(u, u_count);
                              int index_dfa = Get_Index(v, v_count, Mn_num);
                              int index_dfa_u = Get_Index(u, u_count, Mn_num);
                              pMd->node[index_dfa_u][index_dfa].value = Sigma[k];

                              while(( v_count >  2) && (v_count <= Mn_num))
                              {
                              for(int i=0; i<v_count; i++)
                              {for(int j=0; j<Mn_num; j++)
                              {
```

```
                        if(pMn->node[v[i]][j].value == Sigma[k])
                        {
                        x[x_count++] = j;
                        } //if
                        for(int m=0; m < Mn_num; m++)
                        {
                        if(pMn->node[j][m].value == 'e')
                        {
                                if(m != x[x_count-1])
                                {
                                        x[x_count] = m;
                                        x_count++;
                                } //if
                                j = m;
                                m = 0;
                        }//if
                        }//for m
                        } //for j
                         }//for i

                        Sort_Array(x, x_count);
                        int index_dfa_x = Get_Index(x, x_count, Mn_num);
                        pMd->node[index_dfa][index_dfa_x].value = Sigma[k];
                        if((v == x) || (x_count == Mn_num))
                        return;
                        for(int n=0; n<x_count; n++)
                        v[n] = x[n];
                        v_count = x_count;
                        } //while
                        } //if
                }       //for cc
                if(flag == 0)
                        pMd->node[rr+1][0].value = Sigma[k];
        } //for rr
        } //for set_size
} //for k

for(int mm=0; mm < (int)strlen(Sigma); mm++)
        pMd->node[0][0].value= Sigma[mm];

for(int i= 0; i < Md_num; i++)
{
        int j=0;
        int flag1 = 0;
        for(j=0; j< Md_num; j++)
{
```

```
            if(pMd->node[i][j].value != '$')
                    flag1 = 1;

        }
            if(flag1 = = 0)
            {
                            for(int mm=0; mm < (int)strlen(Sigma); mm++)
                                    pMd->node[i][0].value= Sigma[mm];

            }
        }
}

//=================================================================
// Function Func2 is to do five operators for the given extended
// regular expressions
//=================================================================

void Func2(fa* pRes,char op, fa* pL, fa* pR, char *Sigma)
{

        struct fa* pDL = new fa;
        struct fa* pDR = new fa;
        struct fa* pRes1 = new fa;

        switch (op) {
         case 'U':
                Union(pRes, pL, pR);
                break;
         case 'C':
                Concatenation(pRes, pL, pR);
                break;
         case '*':
                Star(pRes, pL);
                break;
         case '!':
                NFA_To_DFA(pRes1, pL, Sigma);
                Complementation(pRes, pRes1);
                break;
         case 'I':
                NFA_To_DFA(pDL, pL, Sigma);
                NFA_To_DFA(pDR, pR, Sigma);
                Intersection(pRes, pDL, pDR);
                break;
         default:
                cout << "An invalid operation!" << endl;
        }
}

//=============================================================
// Function Func1 is to transfer a given string to a "tree" structure,
// then it do the operation on the highest priority operator
//=============================================================

struct fa* Func1 (char* Sigma1, char* sentence, int length)
{
        int count = 0;
        int array[80] = {0};
        char left[80] = {0};
        char right[80] = {0};
        int max_op=0;
        struct fa* result = new fa;
        FA* left_result = new FA();
        FA* right_result = new FA();
        struct fa* Middle = new fa;


        struct fa* F_left = new fa;
        struct fa* F_right = new fa;

        Get_Array(array, sentence);

        int k=0;
        while(k< (int)strlen(sentence))
        {
                bool bSymbol = false;
                bSymbol = sentence[k] == 'U' || sentence[k] == '*'
                                 || sentence[k] == 'I'
                || sentence[k] == 'C' || sentence[k] == '!';
                if (array[k] == 1 && bSymbol)
                {
                Get_Left(left, sentence, k);
                Get_Right(right, sentence, k);
                Transfer_Basic(F_left, Sigma1, left[0]);
                Transfer_Basic(F_right, Sigma1, right[0]);
                Func2(result, sentence[k], F_left, F_right, Sigma1);
                return result;
                }

        else {
                k++;
                }

        }//while
        delete F_left;
        delete F_right;
        return result;
}

//========================================================
// Function Func0 is to transfer the extended regular expressions
// to  a "tree" structure and call the corresponding subroutines
// to do the operation on five operators
//========================================================

void Func0(fa* result0, char* Sigma0, char* sentence0, int length0)
{
        int count = 0;
        int array[80] = {0};
        char left[80] = {0};
        char right[80] = {0};
        int max_op=0;
        FA* left_result = new FA();
        FA* right_result = new FA();
        struct fa* Middle = new fa;

        struct fa* F_left = new fa;
        struct fa* F_right = new fa;


        Get_Array(array, sentence0);
        max_op = Get_Max(array, sentence0);

        int k=0;
        while(k< (int)strlen(sentence0))
        {
                bool bSymbol = false;
                bSymbol = sentence0[k] == 'U' || sentence0[k] == '*'
                                || sentence0[k] == 'I'
                || sentence0[k] == 'C' || sentence0[k] == '!';

                if((array[k] == 1) && bSymbol)
                {
                        Get_Left_Op(left, array, sentence0, k);

                        if(strlen(left) == 1)
                        Transfer_Basic(F_left, Sigma0, left[0]);
```

```
                    else if(strlen(left) <= 5)
                    F_left    Func1(Sigma0, left, length0);
                    else
                    Func0(F_left, Sigma0, left, length0);

                    Get  Right_Op(right, array, sentence0, k);
                     if(strlen(right) == 1)
                            Transfer_Basic(F_right, Sigma0, right[0]);
                    else if(strlen(right) <- 5)
                            F_right = Func1(Sigma0, right, length0);
                    else
                            Func0(F_right, Sigma0, right, length0);

                    Func2(result0, sentence0[k], F_left, F_right, Sigma0);
                    return;

                    }
                else
                    k++;

        }//while

}


//==================================================================
// Function DFS_Visit is to change the color of vertices on the
// path starting from vertex u. It is used for function DFS
//==================================================================

void DFS_Visit(fa* pMM, int u)
{

        int l = pMM->size;
        pMM->node[u][u].color = 'g';

        for(int v = 0; v < l; v++)
                {
                    if(pMM->node[u][v].value != '$')
                    {
                            if(pMM->node[v][v].color == 'w')
                            {
                                    DFS_Visit(pMM, v);
                            }

                    }
                }
        pMM->node[u][u].color = 'b';
}

//===========================================================
// Function DFS is to use depth-first search to solve the
// reachbility problem
//===========================================================

void DFS(fa* pMM)
{
        int l = 1;
        int S = 0;
        int F = 0;
        int len = pMM->size;

        int rr = 0;
        for(rr = 0; rr < len; rr++)
                {
                        pMM->node[rr][rr].color = 'w';
                }

        for(rr = 0; rr < len; rr++)
        {
        if((pMM->node[rr][rr].start == 'Y') && (pMM->node[rr][rr].color == 'w'))
                DFS_Visit(pMM, rr);
        }
}
```

```
                }
        pMM->node[u][u].color = 'b';
}

//==========================================================
// Function DFS is to use depth-first search to solve the
// reachbility problem
//==========================================================

void DFS(fa* pMM)
{
        int l = 1;
        int S = 0;
        int F = 0;
        int len = pMM->size;

        int rr = 0;
        for(rr = 0; rr < len; rr++)
                {
                        pMM->node[rr][rr].color = 'w';
                }

        for(rr = 0; rr < len; rr++)
        {
        if((pMM->node[rr][rr].start == 'Y') && (pMM->node[rr][rr].color == 'w'))
                DFS_Visit(pMM, rr);
        }
}
```

VITA

Min Cai

Candidate for the Degree of

Master of Science

Thesis: THE EMPTINESS PROBLEM FOR EXTENDED REGULAR EXPRESSIONS

Major Field: Computer Science

Biographical:

Education: Graduated from Jinshan High School, Chanzhou, Guangdong, China in July 1993. Received Bachelor of Science degree in Mathematics from Shantou University, Shantou, Guangdong, China in June 1997. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 2002.

Experience: Employed by PULON Computing Company as a programmer and system analyst, Shantou, Guangdong, China in 1997. Employed by Nokia Mobile Phone Company as a part-time programmer, HongKong, March 1999 to May 1999.