AN ALGORITHMIC APPROACH TO CONSTRUCTING FINITE AUTOMATA

by

Charles Louis DeGennaro

In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in

The Department of Computer Science

State University of New York
New Paltz, New York 12561

May 2024
AN ALGORITHMIC APPROACH TO CONSTRUCTING FINITE AUTOMATA

Charles Louis DeGennaro

State University of New York at New Paltz

We, the thesis committee for the above candidate for the Master of Science degree, hereby recommend acceptance of this thesis.

Ashley Suchy, Thesis Advisor
Department of Computer Science, SUNY New Paltz

Chirakkal Easwaran, Thesis Committee Member
Department of Computer Science, SUNY New Paltz

Mike Curry, Thesis Committee Member
Department of Computer Science, SUNY New Paltz

Submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science at the State University of New York at New Paltz
ABSTRACT

In this thesis, we introduce algorithms for creating deterministic (DFA) and nondeterministic finite automata (NFA) for some common types of regular languages.

The motivation for this work comes from studying problems students taking theory of computation classes encounter. When working with finite automata, there are patterns in the structures of automata that arise. This thesis seeks to generalize a subset of common problems by creating step-by-step constructions based on the distinguishing properties of the given language. Along with these algorithms, software has been developed to generate these automata based on their distinguishing properties. This thesis aims to be the foundation for developing an educational tool for theory of computation students. The goal is to highlight the similarities and logic behind these common constructions, while also providing the visual aid of the constructed automata through the accompanying software.

The problems we consider are: the Maximum Prefix-Suffix Overlap problem, the Substring problems, the X followed by Y problem, the Consecutive Character problems, the Conditional Counting Modulo n problems, the Regularly Repeated Characters problem, and the Contains X but not Y problem. All algorithms presented in this thesis are polynomial in runtime, and polynomial in space complexity. The Maximum-Prefix-Suffix Overlap problem has a linear runtime, and the Contains X but not Y problem is of special interest due to its space reduction from the ‘usual’ solution.
Acknowledgments

I would like to start this thesis with a few thank yous, to some very important and influential people that have guided, aided, and supported me through this academic process.

Thank you Ashley, for being an incredible advisor, mentor, and friend throughout my time at New Paltz. From teaching me the basic data structures of computer science, to advising me through my master’s research, I have learned an invaluable amount of lessons through you. When you returned to New Paltz and gave a talk on your research through the interview process, I was just a student of language processing, but I grasped many of the ideas, and became enthralled with the work you were doing. You welcomed me to study with you just for the sake of learning, and we had weekly meetings where you continued to expand my knowledge in the field of formal methods. You led me through undergraduate research, writing my first paper, and attending my first conference. You introduced me to the world of academia, and led me to find my calling. I had many inspirations to pursue this lifestyle, teaching and researching, but seeing the path you took and the encouragement you gave me to follow in your footsteps has led to the most joyous times of my life, and I am forever grateful to be set down this path with you in my corner.

Thank you Professor Dos Reis, for being my introduction to many computer science concepts. My first semester at New Paltz you taught me the fundamentals of programming, and from there my passion for the field skyrocketed. Later you taught me assembly language, the class that I would eventually teach myself, taking heavy inspiration from your teaching style. Finally you taught me language processing, which truly expanded my horizons about what computer science was about. There is the high level programming and web development, but there is also the theoretical and the low level, and through your teachings I found my niche in that area of computer science. Watching you lead a classroom, present a lecture, reading your books, all inspired me to become an educator, to invoke the same feelings in students you invoked in me, a passion for learning and a curiosity about the intricacies of the computer and logic surrounding it.

Thank you Professor Easwaran for providing me with a plethora of opportunities while at New Paltz. You allowed me to step into a graduate student position when I was a sophomore, leading the language processing lab, because you believed in my ability to help students. You listened to my ideas about a general computer science lab, where students can come in for help
in any subject, and allowed me to work and run the lab for a year, where I met many friends and helped countless students in their studies. You accepted me into the masters program and allowed me to be a full time teaching assistant for language processing, letting me share my passion and knowledge with students. Finally you provided me with the greatest opportunity of all, teaching an undergraduate course, while still pursuing my masters. Teaching assembly this past semester has given me a great deal of respect for educators, showing me what it really takes to teach a course, but you believed in my ability to be a teacher, which is the greatest honor to bear.

Thank you to my friends who support and continually make me smile through all the challenges in life. Andrew, andrew, and Trevor, thank you for always being there to listen to whatever crazy ideas or projects I have to share, and grounding me when I get too crazy. Mike and Emily, thank you for your endless amounts of entertainment, UNO games, and to steal a line from Ashley’s dissertation, your taxes. Tom, thank you for inspiring me to improve myself, and continually pushing me to be my best self. To all my other friends I have met through my time at New Paltz, thank you for supporting me and sharing your knowledge with me. Learning, no matter the content, is always a fun experience, and learning through friends is the best way to learn.

Finally, I must thank my family, for their continual and endless support throughout my entire academic journey. Aunts, uncles, grandparents, cousins, you shower me with unconditional love and excitement for my career, and it means the world to have your support in my endeavors. Thank you Grandpa for your nightly calls, telling me whatever jokes you have that night, keeping me connected with family while away at school. I always look forward to a call from you, knowing I am bound to have a wide smile on my face before we say our goodbyes. Thank you Lia for your questionable but undeniable support and love, even if you are confused by my passions. Of course, the greatest thanks goes to my parents, Mom and Dad. Thank you for believing in every endeavor I pursued, and supporting me in more ways than I can keep track of. You may think I’ll never be done with school, but that is my goal. Learning is one of the greatest joys I have found in life, and by keeping myself surrounded with knowledgeable people, I can continue to feed that joy. No matter what I do you continue to push me towards my goals, which seem to continually grow in proportion, with a goalpost that continually gets moved back, but we’re in the home stretch now! Besides your love and support, you have provided me with the greatest gift of all, a set of morals and respect that I couldn’t have learned from anybody else. AATW.
CONTENTS

ABSTRACT ............................................................... ii
Acknowledgments ...................................................... iii
LIST OF FIGURES ....................................................... viii

1. Introduction ......................................................... 1
   1.1 History .......................................................... 1
   1.2 Contributions and Motivation ................................. 2
   1.3 Definitions ..................................................... 3
      1.3.1 Basics ....................................................... 3
      1.3.2 String Basics ............................................... 3
      1.3.3 DFA .......................................................... 4
      1.3.4 NFA .......................................................... 5
   1.4 Notations ....................................................... 5

2. Algorithms .......................................................... 7
   2.1 Maximum Prefix-Suffix Overlap Problem ................. 7
   2.2 The Substring Problems ...................................... 10
      2.2.1 Case 1 – Substring ....................................... 11
      2.2.2 Case 2 – Prefix ........................................... 14
      2.2.3 Case 3 – Suffix ............................................ 16
      2.2.4 Case 4 – Single String ................................. 18
      2.2.5 Single character alphabets ............................ 20
   2.3 The X Followed By Y Problem ............................... 20
   2.4 The Consecutive Characters Problems .................... 25
      2.4.1 Run of character c ..................................... 25
A.3.2 No runs of character $c$ are length $n$ ........................................ 60
A.3.3 All runs of $c$ must be lengths in $M$ ........................................ 62
A.4 The Conditional Counting Modulo n Problem ............................... 64
A.5 The Periodic Character Detection Problem ................................... 65
A.6 The Contains X But Not Y Problem* ........................................... 68
# LIST OF FIGURES

1.1 Formalisms of Regular Languages [17] ................................................. 2

1.2 Example DFA ......................................................................................... 4

2.1 Highlighting the Overlap Problem ........................................................ 7

2.2 The first $p_0$ in $s$ denotes the first time we would start a possible match for $m$, the second $p_0$ denotes where we would restart if $p_i \neq s_j$ ......................................................... 8

2.3 A demonstration of $s$ jumping back to the first occurrence of $p_0$ in $s$ from it’s last starting position when some $p_i \neq s_{k-i-1}$. $m$ is the current state of $m$ before the comparison is made ................................................................. 8

2.4 Substring NFA, $w = aabbab$ ............................................................... 11

2.5 Substring DFA, $w = aabbab$ ............................................................... 13

2.6 Prefix NFA, $w = aabbab$ ................................................................. 14

2.7 Prefix DFA, $w = aabbab$ ................................................................. 15

2.8 Suffix NFA, $w = aabbab$ ................................................................. 16

2.9 Suffix DFA, $w = aabbab$ ................................................................. 18

2.10 Trivial NFA, $w = aabbab$ ............................................................... 19

2.11 Trivial DFA, $w = aabbab$ ............................................................... 19

2.12 $x$ followed by $y$ NFA, $x = abb$, $y = ba$ ........................................ 22

2.13 $x$ followed by $y$ DFA, $x = abb$, $y = ba$ ........................................ 23

2.14 No occurrences of $x$ DFA, $x = bab$ .................................................... 24

2.15 Run of $c$ DFA, $c = a$ ................................................................. 26

2.16 Runs of $c$ must be length $n$ NFA, $c = a$, $n = 4$ ............................. 27
2.17  Only runs of $c$ with length $n$ DFA, $c = a$, $n = 4$ ........... 28

2.18  No runs of $c$ with length $n$ NFA, $c = a$, $n = 4$ ........... 29

2.19  No runs of $c$ with length $n$ DFA, $c = a$, $n = 4$ ........... 30

2.20  Only runs of $c$ with lengths in $M$ NFA, $c = a$, $M = \{2,4\}$ ........... 32

2.21  Only runs of $c$ with lengths in $M$ DFA, $c = a$, $M = \{2,4\}$ ........... 33

2.22  Modulo Construction, $x = 4$, $y = 2$, $A = \{a\}$, $c = (=)$ ........... 35

2.23  Regularly repeating characters NFA, $x = 4$, $y = 2$, $A = \{a\}$ ........... 37

2.24  Regularly repeating characters DFA, $x = 4$, $y = 2$, $A = \{a\}$ ........... 38

2.25  Contains $x$ but not $y$ DFA, $x = aaba$, $y = aababa$ ........... 40
CHAPTER 1
Introduction

In this chapter, we will first briefly examine the history surrounding the research to be discussed in the rest of this thesis. Next, we will see an overview of the main contributions made in this thesis, before we define key concepts used in the remaining chapters.

1.1 History

Languages have been studied using two different formalisms, grammars and automata. Grammars and automata are equivalent ways of representing languages. A grammar is used to produce or derive a set of strings that form the language by using a set of rewrite rules. An automaton accepts the set of strings that form the language by following the transition function defined for the abstract machine. The Chomsky hierarchy demonstrates languages from regular, the simplest, to recursively enumerable, the most complex.

This thesis will focus on the regular languages. Commonly known formalisms of regular languages are deterministic finite automata (DFA), nondeterministic finite automata (NFA), regular expressions, and regular grammars. In this thesis, we will focus on DFAs and NFAs. Figure 1.1 shows the regular language formalisms. Note that all these different ways of capturing the regular languages are equivalent, and there are well-known algorithms to convert between the different formalisms.

There are many existing algorithms to work with regular languages. Some algorithms given in [16], [10] are: converting an NFA to a DFA, converting between an NFA and a regular expression, minimizing a DFA, and converting between NFAs and right-linear grammars. Some lesser-known algorithms have been introduced in [5], [13], [17]: converting a prefix grammar to a right-linear grammar, converting a DFA to a prefix grammar, converting a suffix grammar to a left-linear grammar, converting a left-linear phrase-structure grammar to a left-linear grammar, and converting a right-linear phrase-structure grammar to a right-linear grammar. There are also closure algorithms that exist for regular languages which allow us to combine multiple finite automata together. Some of the operations that closure algorithms exist for are union, intersection, reverse,
concatenation, complement, difference, right-quotient, left-quotient, Kleene star [16],[10].

![Diagram of Formalisms of Regular Languages](image)

Figure 1.1: Formalisms of Regular Languages [17]

1.2 Contributions and Motivation

The motivation for this research was interest in developing more algorithms for the regular languages and in developing tools to help students learn theory of computation topics more easily.

Most of the work presented consists of constructions for both NFAs and DFAs that generalize common problems an introductory Theory of Computation student might encounter. This generalization takes the form of defining languages with arguments one could set themselves, and based on those input parameters the algorithms presented will construct the appropriate automaton for that language. The two main goals when working on this thesis were to satisfy my curiosity in generalizing these problems, as well as to act as a guide for students new to the field.

The following problems are introduced and algorithms to answer them are given in the remaining chapters of this thesis. Some of the problems with polynomial-time algorithms to be discussed are:

- finding the maximum overlap between the prefix of a string and the suffix of another string, the Maximum Prefix-Suffix Overlap problem
- building FAs for languages that contain a specific substring, the Substring problems
- building FAs for languages where every occurrence of a string X is directly followed by an occurrence of a string Y, the X followed by Y problem
- building FAs for languages that have runs of characters meeting certain length requirements, the Consecutive Characters problems
• building FAs for languages where there is a certain amount modulo some n number of characters in each string, the Conditional Counting Modulo n problems

• building FAs for languages where a certain character appears at every $n^{th}$ position in the string, the Periodic Character Detection problem

• building FAs for the languages that contain the substring X but not the substring Y, the Contains X but not Y problem.

1.3 Definitions

We will now introduce some basic definitions and notations that will be used throughout the paper, to give the proper background information required to understand the topics discussed. For more information regarding topics such as DFAs, NFAs, we refer the reader to [16].

1.3.1 Basics

Below is a brief introduction to key components within the field of formal languages.

An alphabet is a set of symbols, denoted by $\Sigma$.

A language is a set of strings. denoted by the symbol $\mathcal{L}$.

The universal set, $\Sigma^*$ is the set of all possible strings over the alphabet $\Sigma$.

In this thesis, we will only be considering the set of regular languages. A regular language is a language that can be recognized by some finite automaton.

1.3.2 String Basics

The empty string is denoted by $\lambda$.

A substring is any continuous section of a string, including the entire string itself.

A prefix is any substring that includes the beginning of the string. Ex: All the prefixes of Emily are $\lambda, E, Em, Emi, Emil, Emily$.

A suffix is any substring that includes the end of the string. Ex: All the suffixes of Mike are Mike, ike, ke, e, $\lambda$. 
String Indexing: To denote specific characters in a string, we will include a subscript on the string symbol with the desired index of our character. Strings will be 0-indexed. For example, if \(s = abcd e\), then \(s_2 = c\).

String Length: To denote the length of a string \(s\), we will use \(|s|\). For example, if \(s = abcd e\), then \(|s| = 5\).

String Concatenation: To concatenate two strings together, or a string and a character, we simply write them next to each other. For example, if \(t = Tom\), \(m = Matters\), and \(a = tm\), then \(a = TomMatters\). Similarly, we can concatenate a string and a character such as \(s = ab\), then \(sa = abba\).

1.3.3 DFA

A deterministic finite automaton (DFA) is a structure made of states and transitions, where each state must have exactly one transition out for each character in the given alphabet.

A DFA is a tuple \((Q, \Sigma, \delta, q_0, F)\), where

- \(Q\) is a finite set of states
- \(\Sigma\) is a finite alphabet
- \(\delta\) is the transition function, \(\delta : Q \times \Sigma \rightarrow Q\), where every state has exactly one transition defined for each symbol in \(\Sigma\)
- \(q_0 \in Q\) is the initial or start state
- \(F \subseteq Q\) is the set of final or accept states

Note the number of transitions in a DFA is exactly \(|Q| \times |\Sigma|\).

[Diagram of a DFA]

Figure 1.2: Example DFA
The initial state is usually denoted by a state with an arrow coming into it with no source state, see state $q_0$ in Figure 1.2.

An accepting state is usually denoted by a state with an extra circle in its border, see state $q_1$ in Figure 1.2.

A dead state is a state that transitions to itself over $\Sigma$ and is non-accepting, essentially failing any input string that enters, see state $q_{rip}$ in Figure 1.2.

States: States have a similar notation to string indexing, but we reserve the symbol $q$ to represent states, and subscripts on $q$ represent a specific state. There are some cases where a different symbol represents a state, but a string and state will not be represented with the same symbol in any given construction. For example, $q_0$ is a state. $q_{|s|}$ is also a state, which is given by the length of $s$.

We say a string is in the language or accepted by a DFA if when finished reading the string the machine stops in an accepting state.

### 1.3.4 NFA

A nondeterministic finite automaton is a structure made of states and transitions, where each state may have any number of transitions out for each character in the given alphabet, including no transition for some characters. An NFA also allows for transitions on the empty string.

An NFA is a tuple $(Q, \Sigma, \delta, q_0, F)$, where the only difference from DFA definition is in the transition function

- $\delta$ is the transition function, $\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$, where every state can have zero or more transitions defined for each symbol in $\Sigma \cup \{\lambda\}$

### 1.4 Notations

Below is an overview of the notations used throughout the paper.

**Range:** When writing out a range, it will take the form of “[start, end]”, with both the start and the end index being included. For example, [1,3] denotes the elements 1, 2, and 3.

**Transitions:** When writing out transitions in algorithms, we will use $q_i \xrightarrow{a} q_j$ where $q_i, q_j \in$
\( Q \) and \( a \in \Sigma \cup \{ \lambda \} \). If we have multiple characters we would like to transition over, it will be represented as a set of those characters. For example, if \( q_i \) transitions to \( q_j \) over the characters \( a \) and \( b \), it will be represented as: \( q_i \xrightarrow{\{a,b\}} q_j \). This corresponds to our usual notion of having \( q_i \xrightarrow{a,b} q_j \), and the set notation is just a computation aid in the algorithms. If we want to transition over the whole alphabet we use \( q_i \xrightarrow{\Sigma} q_j \). For any construction, if the transition is a set subtraction that results in the empty set, then the construction does not add the transition.

\[
\begin{array}{c}
\text{Min: An operation that will return the minimum value of all the parameters passed. For example, Min}(3, 1) = 1. \text{ Note that more than 2 elements or a set of values can be passed to the Min function.} \\
\text{Max: An operation that will return the maximum value of all the parameters passed. For example, Max}(3, 1) = 3. \text{ Note that more than 2 elements or a set of values can be passed to the Max function.} \\
\text{Substring: When defining a language, we will use the “in” symbol (\( \in \)) from set notation to define some string must be a substring of another. For example, if w = Cat, and x \( \in \) w, then x must be one of the elements from the following set of strings: \{\lambda, C, a, t, Ca, at, Cat\}.} \\
\text{Prefix: When defining a language, we will use the “in” symbol (\( \in_p \)) from set notation with a subscript of “p” to define some string must be a prefix of another. For example, if w = Cat, and x \( \in_p \) w, then x must be one of the elements from the following set of strings: \{\lambda, C, Ca, Cat\}.}
\end{array}
\]
CHAPTER 2

Algorithms

This chapter introduces problems and provides constructive algorithms for the problems introduced. The first problem to be discussed is the Maximum Prefix-Suffix Overlap problem. The solution to this problem is used as an intermediary step for subsequent algorithms in this chapter. The remaining problems all have algorithms given for building both a DFA and NFA to solve them.

2.1 Maximum Prefix-Suffix Overlap Problem

Before looking into the automata constructions, we look at a sub-problem that has arisen as a step in multiple constructions. Given two strings, \( p \) and \( s \), what is the longest string match \( m \) between a prefix of \( p \) and a suffix of \( s \). In other words, \( p = mp' \), \( s = s'm \). Note that \( p', s' \), or both could be the empty string.

Note this is similar to the ‘usual’ string matching problem where we want to find a pattern match in a text. This problem can be solved in linear time using the Knuth-Morris-Pratt algorithm [4].

Let \( p, s \) be two strings where \( |p| = n \) and \( |s| = k \) then the maximum prefix-suffix overlap is shown in Figure 2.1, and \( m = p_0p_1p_2 \ldots p_i = s_{k-1-i} \ldots s_{k-3} s_{k-2} s_{k-1} \), where \( p_0 = s_{k-1-i} \) and so on up to \( p_i = s_{k-1} \), where \( 0 \leq i \leq \text{Min}(|p|, |s|) \). The box is a visual aid to see the overlap. For example, with the strings, \( p = abba \) and \( s = abab \), then the maximum overlap \( m \) is \( ab \), which is underlined.

\[
\begin{array}{cccccccc}
 s_0 & s_1 & s_2 & \cdots & s_{k-1-i} & \cdots & s_{k-3} & s_{k-2} & s_{k-1} \\
 p_0 & p_1 & p_2 & \cdots & p_i & \cdots & p_{n-3} & p_{n-2} & p_{n-1}
\end{array}
\]

Figure 2.1: Highlighting the Overlap Problem

The overlap string \( m \) is clearly unique, and \( 0 \leq |m| \leq \text{Min}(|p|, |s|) \). If \( |p| \leq |s| \), then we have an upper bound of \( |p| \), and we are restricted to looking at the last \( |p| \) symbols of \( s \). Similarly if \( |s| \leq |p| \) we are restricted to the first \( |s| \) symbols of \( p \). Hence the problem is a ‘localized’ matching problem given two fixed points. This can be done in linear time with respect to \( \text{Min}(|p|, |s|) \).
The average runtime is linear $\text{Min}(|p|, |s|)$. The best case runtime is also $\text{Min}(|p|, |s|)$ and one example of the best case occurs when $s = wp$ ($s$ is a suffix of $p$). For example, when $p = abbb$ and $s = wabbb$ where $w$ can be any string we will only need to make $|p|$ data comparisons. The worst case runtime is $2 \times \text{Min}(|p|, |s|)$ and one example of the worst case occurs when $p = aw$ where $a$ is any character and $w$ is a string with no occurrences of $a$, and $s$ is a string consisting of only $a$’s. For example, when $p = abbb$ and $s = aaaa$ we will need to make two data comparisons for every symbol in $p$.

![Figure 2.2: The first $p_0$ in $s$ denotes the first time we would start a possible match for $m$, the second $p_0$ denotes where we would restart if $p_i \neq s_j$](image)

<table>
<thead>
<tr>
<th>$s$</th>
<th>$p$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 4: a a b [ ] a a a b [ ] a m=abb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 5: a [ ] b a a [ ] a a b b a m=λ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step 6: a a [ ] b a a a [ ] b b a m=a</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 2.3: A demonstration of $s$ jumping back to the first occurrence of $p_0$ in $s$ from it’s last starting position when some $p_i \neq s_{k-i-1}$. $m$ is the current state of $m$ before the comparison is made](image)

Note that for readability in Algorithm 1, string indexing is done by putting the index in square brackets after the string, instead of as a subscript. Ex: $s = abcde$, $s[2] = c$. 

8
Algorithm 1 Maximum Prefix-Suffix Overlap Problem

Let $p$, $s$ be two strings

1. Let $s_{\text{index}} = \text{Max}(0, |s| - |p|)$. Note $s_{\text{index}} = 0$ if $|s| \leq |p|$, and $s_{\text{index}} = |s| - |p|$ otherwise. $s_{\text{index}}$ will be the index in the string $s$ where the search for $m$ will begin.

2. Let $p_{\text{index}} = 0$. The search for $m$ will always start from index 0 in $p$.

3. Let $m = \lambda$.

4. While $p_{\text{index}} < |p|$ and $s_{\text{index}} < |s|$

   4a. If $s[s_{\text{index}}] = p[p_{\text{index}}]$ then $p_{\text{index}} = p_{\text{index}} + 1$, $s_{\text{index}} = s_{\text{index}} + 1$, $m = m + s[p_{\text{index}}]$.

   If the current symbols match then move to the next symbol in each string and add the current symbol to $m$ and continue the search.

   4b. If $s[s_{\text{index}}] \neq p[p_{\text{index}}]$ and there has been another occurrence of $p_0$ found at index $y$ in $s$ then $p_{\text{index}} = 0$, $s_{\text{index}} = y$, $m$ is reset to the empty string.

   Reset the search for $m$ from the last ‘unsearched’ occurrence of $p_0$. In the first search for $m$ we start the matching from the very first occurrence of $p_0$ in $s$. Now if we have encountered another occurrence of $p_0$ in $s$ at this moment we can reset from that point. See Figure 2.2. Let $x$ be the index we started this match from in $s$. It is the case that all symbols in $p$ from index 0 to $p_{\text{index}} - 1$ matched the symbols in $s$ from the first occurrence of $p_0$ to index $s_{\text{index}} - 1$ matched, but the symbols at index $p_{\text{index}}$ in $p$ and $s_{\text{index}}$ in $s$ don’t match. We know the next possible $m$ starts at the second occurrence of $p_0$ in $s$, let this index by $y$. Therefore, $p_{\text{index}} = 0$, $s_{\text{index}} = y$, $m$ is reset to the empty string.

   4c. If $s[s_{\text{index}}] \neq p[p_{\text{index}}]$ and there has not been another occurrence of $p_0$ found then $p_{\text{index}} = 0$, $s_{\text{index}} = s_{\text{index}} + 1$, $m$ is reset to the empty string.

   At this point we know the maximal length of $m$ must be one less hence we move ahead in $s$, but we need to go back to the first symbol of $p$, and we restart the search for $m$. 
2.2 The Substring Problems

A common computational problem is that given a string \( w \in \Sigma^* \), build a finite automaton for the regular language \( \mathcal{L} \) where every string in the language contains \( w \) as a substring.

Here we present constructions for both an NFA and DFA, with four cases to consider when solving this problem.

First, we’ll define the form of our strings. Let \( z \in \Sigma^* \) and \( w \) be the substring. We can decompose \( z \) into three components: \( x \), which is the substring before \( w \), \( w \) itself, and \( y \), the substring after our \( w \). More simply, \( z = xwy \). Note that \( x \), \( y \), or \( x \) and \( y \) simultaneously could be \( \lambda \).

We can now say that our language will consist of strings that have the following format:

\[ \mathcal{L} = \{ xwy \mid x, y \in \Sigma^* \} \]

Before going into the constructions, we should note a couple of properties that will be present in each construction in this section.

**Property 1**: Each state is associated with a character in the substring. For example, the state \( q_2 \) will have a transition to state \( q_3 \) on the character associated with its state number, \( w_2 \). In general, state \( q_i \) will transition to state \( q_{i+1} \) on character \( w_i \).

**Property 2**: Each state represents the length of our substring \( w \) detected at any point in processing some input string. For example, if we are processing some input string and are in state \( q_3 \), we can say that the last 3 characters read in from our input string are the first 3 characters of our substring \( w \).
There are four cases to consider:

Case 1: \( x \neq \lambda, \ y \neq \lambda \)

Case 2: \( x = \lambda, \ y \neq \lambda \)

Case 3: \( x \neq \lambda, \ y = \lambda \)

Case 4: \( x = \lambda, \ y = \lambda \)

Note that case 1 corresponds to \( w \) occurring as a substring anywhere in each string in the language. Case 2 corresponds to \( w \) occurring exclusively as a prefix of every string in the language and case 3 corresponds to \( w \) occurring exclusively as a suffix of every string in the language. Case 4 is the language containing a single string, namely \( w \).

2.2.1 Case 1 – Substring

When both \( x \) and \( y \) are not empty, we have the most general case of our problem. For our NFA, we start by creating \( |w| + 1 \) states labeled \( q_0 \) through \( q_{|w|} \), with \( q_0 \) as our starting state. Then following Property 1, we connect each state in sequence based on the character at that index in our input string \( w \). Formally, for all \( i \) in \([0, |w| - 1]\), create the transition \( q_i \) to \( q_{i+1} \) over \( w_i \). We make \( q_{|w|} \) the final state, as once we get one occurrence of \( w \), we should always accept. Next we add a self-loop over \( \Sigma \) on \( q_0 \), to ensure we are always in \( q_0 \) to detect an occurrence of \( w \). Finally we add a self-loop over \( \Sigma \) on \( q_{|w|} \), so that once we detect an occurrence of \( w \), we always stay in the final accepting state.

![Substring NFA](image)

Figure 2.4: Substring NFA, \( w = aabbab \)
Algorithm 2 Substring NFA
Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{xwy \mid x, y \in \Sigma^*, x, y \neq \lambda\} \)

Steps for NFA:

1. Create \( |w| + 1 \) states, labeled \( q_0 \ldots q_{|w|} \)
2. Make \( q_0 \) the starting state
3. Make \( q_{|w|} \) the final state
4. For all \( i \) in \([0, |w| - 1]\), create the transition \( q_i \xrightarrow{w_i} q_{i+1} \)
5. Add self-loop transition \( q_0 \xrightarrow{\Sigma} q_0 \)
6. Add self-loop transition \( q_{|w|} \xrightarrow{\Sigma} q_{|w|} \)

The DFA for this problem is a little trickier to generalize, however, once the pattern is spotted the construction is trivial. We once again start by creating \( |w| + 1 \) states labeled \( q_0 \) through \( q_{|w|} \), with \( q_0 \) as our starting state, and connecting them together as described by Property 1. Then we define a substring of \( w \) called \( p \), which is the longest prefix of \( w \) that only contains the character \( w_0 \). For example, if \( w = aaabab \), then \( p = aaa \). Now we can break the problem into four separate categories of states each with their own unique properties:

- The Prefix States \( (q_0 - q_{|p| - 1}) \)
- End of Prefix State \( (q_{|p|}) \)
- Post Prefix States \( (q_{|p| + 1} - q_{|w| - 1}) \)
- The Final State \( (q_{|w|}) \)

The Prefix States will all transition from themselves to \( q_0 \) over \( \Sigma - \{w_0\} \), since they all share their transition of \( w_0 \) to the next state as described by Property 1.

The End of Prefix State will contain a self-loop over \( w_0 \), since if we continue to get this character, we can’t continue to check further characters in \( w \), but can ensure the last \(|p|\) characters...
read in were \( w_0 \) from Property 2. Due to this state marking the end of the prefix, we also know that \( w_{|p|} \neq w_0 \), so all other characters encountered will reset progress on checking the substring, leading to a transition from \( q_{|p|} \) to \( q_0 \) over \( \Sigma - \{w_0, w_{|p|}\} \).

The Post Prefix States have a maximum of three transitions they can make:

- A transition to \( q_{i+1} \) as described by Property 1
- A transition to \( q_1 \) over \( \{w_0\} - \{w_i\} \)
- A transition to \( q_0 \) over \( \Sigma - \{w_i, w_0\} \), where \( i \) is the current state \( q_i \)

Skipping the transition described by Property 1 since that has been discussed, we can make the transition to \( q_1 \) over \( w_0 \), because \( q_1 \) represents when we detected the first character of \( w \) as stated in Property 2. However we do not add the transition if \( w_i \) is equal to \( w_0 \), because \( w_i \) is already on the transition to \( q_{i+1} \). The final transition to \( q_0 \) handles all other characters, \( \Sigma - \{w_i, w_0\} \), which forces us to restart our search for the substring.

Finally, our final state is only reachable once the entire substring has been continuously detected, at which point the rest of the input will always be accepted, leading to \( q_{|w|} \) being set as an accepting state, and a self-loop of \( \Sigma \) on \( q_{|w|} \).

![Substring DFA Diagram](image)

Figure 2.5: Substring DFA, \( w = aabbab \)
Algorithm 3 Substring DFA

Let $w \in \Sigma^*$

Let $\mathcal{L} = \{xwy \mid x,y \in \Sigma^*, x,y \neq \lambda\}$

Steps for DFA:

1. Create $|w| + 1$ states, labeled $q_0 \ldots q_{|w|}$

2. Make $q_0$ the starting state

3. Make $q_{|w|}$ the final state

4. For all $i$ in $[0, |w| - 1]$, create the transitions $q_i \xrightarrow{w_i} q_{i+1}$

5. Add self-loop transition $q_{|w|} \xrightarrow{\Sigma} q_{|w|}$

6. Let $p$ be the longest prefix of $w$ consisting only of the character $w_0$

7. Add self-loop transition $q_{|p|} \xrightarrow{w_0} q_{|p|}$

8. For all $i$ in $[|p| + 1, |w| - 1]$ add transitions $q_i \xrightarrow{\{w_0\} - \{w_i\}} q_1$

9. For all $i$ in $[0, |w| - 1]$ add transitions $q_i \xrightarrow{\Sigma - \{w_0, w_i\}} q_0$

2.2.2 Case 2 – Prefix

To find only strings where our substring $w$ is the prefix of the string, $x$ would be $\lambda$. For this reason, case 2 will cover Prefix Constructions. The NFA for this case is nearly identical to the one presented in Case 1, with the only difference being that there would be no self-loop on $q_0$, since we want to ensure that $w$ is the very beginning of any string passed into this automaton.

![Figure 2.6: Prefix NFA, $w = aabbab$](image)
Algorithm 4 Prefix NFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{ wy | y \in \Sigma^*, y \neq \lambda \} \)

Steps for NFA:

1. Create \(|w| + 1\) states, labeled \( q_0 \ldots q_{|w|} \)
2. Make \( q_0 \) the starting state
3. Make \( q_{|w|} \) the final state
4. For all \( i \) in \([0, |w| - 1]\), create the transition \( q_i \xrightarrow{w_i} q_{i+1} \)
5. Add self-loop transition \( q_{|w|} \xrightarrow{\Sigma} q_{|w|} \)

To construct our DFA, we can use the NFA construction above as a base, as all we need to add is the dead state \( q_{rip} \) and transition to it over every character not described by Property 1. In particular, the transitions we need to add are from every state \( q_i \) to \( q_{rip} \) over \( \Sigma - \{w_i\} \), where \( i \) is in the range \([0, |w| - 1]\), along with the self-loop transition to \( q_{rip} \) over \( \Sigma \).

![Figure 2.7: Prefix DFA, \( w = aabbab \)](image-url)

Figure 2.7: Prefix DFA, \( w = aabbab \)
Algorithm 5 Prefix DFA

Let $w \in \Sigma^*$

Let $\mathcal{L} = \{wy \mid y \in \Sigma^*, y \neq \lambda\}$

Steps for DFA:

1. Use the construction from Algorithm 4 with $w = w$

2. Create a state $q_{rip}$

3. For all $i$ in $[0, |w| - 1]$, create the transition $q_i \xrightarrow{\Sigma - \{w_i\}} q_{rip}$

4. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$

2.2.3 Case 3 – Suffix

To find the strings where $w$ is exclusively the suffix of our string, $y$ must be $\lambda$. These next algorithms for that reason will cover Suffix Constructions. The NFA for this case is once again similar to the one presented in Case 1, but this time we remove the self-loop on the final state $q_{|w|}$, since we don’t want to stay in the final state if we simply detect $w$, we must end our input string with $w$.

![Figure 2.8: Suffix NFA, $w = aabbab$](image-url)
Algorithm 6 Suffix NFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{xw \mid x \in \Sigma^*, x \neq \lambda \} \)

Steps for NFA:

1. Create \(|w| + 1\) states, labeled \( q_0 \ldots q_{|w|} \)

2. Make \( q_0 \) the starting state

3. Make \( q_{|w|} \) the final state

4. For all \( i \in [0, |w| - 1] \), create the transition \( q_i \xrightarrow{w_i} q_{i+1} \)

5. Add self-loop transition \( q_0 \xrightarrow{\Sigma} q_0 \)

The process in creating the DFA is very similar to the general case, as the only state that is different is the final state \( q_{|w|} \). Instead of having a self-loop, it must transition to a previous state (or itself for \( w \)’s made of only a single character), with each character leading to a different state based on overlap characteristics. To prepare for this, we first start with the DFA construction of Algorithm 3, but remove the final state on \( q_{|w|} \).

For every character \( c \in \Sigma \), we need to determine these overlap characteristics to make our transitions out of \( q_{|w|} \). This overlap string is defined by Algorithm 1, where the prefix string given is \( w \), and the suffix string is \( w \) with the character we want to transition over concatenated at the end, \( wc \). We will call this output string \( m \), and add the transition from \( q_{|w|} \) to \( q_{|m|} \) over \( c \). For example, if \( m = ab \), we would make the transition of \( q_{|w|} \) to \( q_2 \) over \( c \). This overlap lets us determine how much of our substring is detected if we get another character after finding our substring, and from Property 2 we know that the state we end up in must have detected \(|m|\) characters of \( w \) upon entering it.
### Algorithm 7 Suffix DFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{xw \mid x \in \Sigma^*, x \neq \lambda \} \)

**Steps for DFA:**

1. Use the construction from Algorithm 3 with \( w = w \)

2. Remove the self-loop transition on \( q_{|w|} \)

3. For all \( c \in \Sigma \), let \( m_c \) be the output from Algorithm 1 where the prefix \( p = w \) and the suffix \( s = wc \)

4. Create the transition \( q_{|w|} \xrightarrow{c} q_{|m_c|} \) for all \( c \in \Sigma \)

---

### 2.2.4 Case 4 – Single String

The last case is trivial, as when both \( x \) and \( y \) are \( \lambda \), the machine should only accept \( w \) itself. For the NFA, this is the same as Case 1, except there are no self-loops on \( q_0 \) or \( q_{|w|} \), since we only want to accept the string \( w \) itself.
Algorithm 8 Trivial NFA
Let \( w \in \Sigma^* \)
Let \( \mathcal{L} = \{w\} \)

Steps for NFA:

1. Create \(|w| + 1\) states, labeled \(q_0 \ldots q_{|w|}\)

2. Make \(q_0\) the starting state

3. Make \(q_{|w|}\) the final state

4. For all \(i\) in \([0, |w| - 1]\), create the transition \(q_i \xrightarrow{w_i} q_{i+1}\)

The DFA is also straightforward, as it is the same as the DFA from Case 2, except there is no self-loop on the final state, with instead the transition from \(q_{|w|}\) going to \(q_{rip}\) over \(\Sigma\).
Algorithm 9 Trivial DFA
Let $w \in \Sigma^*$
Let $\mathcal{L} = \{w\}$

Steps for DFA:

1. Use the construction from Algorithm 8 with $w = w$
2. Create a state $q_{rip}$
3. For all $i$ in $[0, |w| − 1]$ create the transition $q_i \xrightarrow{\Sigma - \{w_i\}} q_{rip}$
4. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$
5. Create the transition $q_{|w|} \xrightarrow{\Sigma} q_{rip}$

2.2.5 Single character alphabets

When our alphabet is just a single character, the meaning of these automaton shift slightly from their original purpose. For example, the substring problem boils down to checking if a given string is greater to or equal in length to string $w$. The prefix and suffix problems lead to the same result. The trivial solution still holds, as it just checks that the only string in the language is $w$ itself.

2.3 The X Followed By Y Problem

An interesting problem to look at is when we have two strings, $x, y \in \Sigma^*$, where every occurrence of $x$ is followed directly by $y$. The general solution is not too hard to construct, borrowing many ideas from the substring constructions above, however there are two conditions which result in an interesting interpretation of the question.

We can define the language of this problem as all strings where the number of occurrences of string $x$ is equal to the number of times $xy$ is also in the string. If these two counts are not equal, then we must have detected $x$ alone at some point, without $y$ following it. Formally, we can define the language as $\mathcal{L} = \{w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w\}$. 

20
Starting with the general solution, our first step is to detect when we find an occurrence of $x$, because then we have to start tracing the characters of $y$. To do this, we can use the DFA construction of the substring problem given by Algorithm 3. However just applying this algorithm on $x$ itself would have the final state represent where we should start detecting $y$. For clarity in constructing the portion of the machine that checks for $y$, we choose to omit this final state, by instead applying Algorithm 3 on all of $x$ except the last character. To be precise, we define a prefix of $x$, $x'$ where $x' = x_0 \ldots x_{|x| - 2}$. We then use Algorithm 3 with $w = x'$.

The subsection of the machine that detects $y$ can be modeled with Property 1, in which we create $|y|$ states labeled $r_0 \ldots r_{|y| - 1}$, and then add the transitions from $r_i$ to $r_{i+1}$ over $y_i$ for $0 \leq i \leq |y| - 2$. We omit the last character of $y$ from this application of Property 1 because after the final character in $y$, we return to detecting an occurrence of $x$. To know what state we should return to in the $x$ detecting section, we need to determine the maximum overlap between the suffix of $y$ and the prefix of $x$. Using Algorithm 1 with $p = x$ and $s = y$, we save the resulting string as $m$. Using a similar technique as in steps 3 and 4 from Algorithm 7, we can add a transition from $r_{|y| - 1}$ to $q_{|m|}$ over $y_{|y| - 1}$.

One of the final steps is to add the transitions for $q_{|x| - 1}$. We start by removing the self-loop transition over $q_{|x| - 1}$, since we no longer want to always accept if we find the substring $x'$. Over the last character in $x$, $x_{|x| - 1}$ we should transition to the beginning of our $y$ detecting subsection, which is state $r_0$. Then, similar to Steps 8 and 9 in Algorithm 3 we will transition to either $q_0$ or $q_1$ based on the remaining characters in $\Sigma$.

Finally, we make all of the $q$ states final ($q_0 - q_{|x'|}$), as we should always accept unless we detect $x$, where we then need to check that the following string of characters is $y$.

The resulting automaton from this construction is the NFA that describes our $x$ followed by $y$ problem.
Figure 2.12: $x$ followed by $y$ NFA, $x = abb$, $y = ba$

**Algorithm 10** $x$ followed by $y$ NFA

Let $x, y \in \Sigma^*$

Let $\mathcal{L} = \{ w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w \}$

**Steps for NFA:**

1. Let $x'$ be $x_0 \ldots x_{|x| - 2}$

2. Use the construction from Algorithm 3 with $w = x'$

3. Remove self-loop on $q_{|x'|}$

4. Make states $q_0 \ldots q_{|x'|}$ final

5. Add $|y|$ states labeled $r_0 \ldots r_{|y| - 1}$

6. Add a transition from $q_{|x| - 1} \xrightarrow{x_0\ldots x_{|x| - 1}} r_0$

7. Add the transitions $q_{|x| - 1} \xrightarrow{\{x_0\} - \{x_0\}_{|x| - 1}} q_1$ and $q_{|x| - 1} \xrightarrow{\Sigma - \{x_0, x_{|x| - 1}\}} q_0$

8. For all $i$ in $[0, |y| - 2]$, add transitions $r_i \xrightarrow{y_i} r_{i+1}$.

9. Let $m$ be the output from Algorithm 1 where the prefix $p = x$ and the suffix $s = y$

10. Create a transition $r_{|y| - 1} \xrightarrow{y_{|y| - 1}} q_{|m|}$
The DFA of the above automaton is very simple, as the only missing transitions are from the \( r \) states. If during our check for \( y \), we encounter a character that would cause \( y \) to not be following \( x \), then we know that all instances of \( x \) are not followed by \( y \), and we should always reject the string at this point. To facilitate this we create a dead state \( q_{rip} \), and then add the transitions \( r_i \) to \( q_{rip} \) over \( \Sigma - \{y_i\} \) for all \( 0 \leq i \leq |y| - 1 \). Finally we add the self-loop on \( q_{rip} \) over \( \Sigma \). Now we have the DFA that describes the general case of the above problem.

![Figure 2.13: x followed by y DFA, \( x = abb \), \( y = ba \)](image)

**Algorithm 11 x followed by y DFA**

Let \( x, y \in \Sigma^* \)

Let \( \mathcal{L} = \{ w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w \} \)

**Steps for DFA:**

1. Use the NFA construction from Algorithm 10 with \( x = x \) and \( y = y \).

2. Add a state \( q_{rip} \)

3. Add self-loop transition \( q_{rip} \xrightarrow{\Sigma} q_{rip} \)

4. For all \( i \) in \([0, \ |y| - 1]\), add transitions \( m_i \xrightarrow{\Sigma - \{y_i\}} q_{rip} \)
An interesting case occurs when $x$ is a substring of $y$. After encountering the first occurrence of $x$, we must start detecting for $y$, however, in detecting $y$, we encounter $x$ again, so what should the machine do? The interpretation here is that after the first occurrence of $x$, the machine can never accept. The reasoning for this is that every occurrence of $x$ must be followed by $y$, thus after our first occurrence of $x$, when we look for $y$ we will find $x$ again, and once again have to start detecting $y$. The solution in this case is to only accept when we have no occurrences of $x$, because if we don’t detect $x$, then the rule still holds that all occurrences of $x$ are followed by $y$. The NFA and DFA for this machine are essentially the same, simply being the complement of Algorithm 3 where $w = x$. To reduce the NFA by one state you could simply remove the dead state $q_{|x|}$ and all transitions associated it.

Another interesting case occurs when $x$ is the suffix of $xy$. We once again run into a similar scenario as before, as once we find our first occurrence of $x$, after completing $y$ we will find that $x$ has been detected again, and we are stuck in another infinite loop of detecting $x$ and following it with $y$. Due to this infinite loop of never accepting, we can only accept before the first occurrence of $x$. Therefore, this also results in the same NFA/DFA is the above case, leading to these two edge cases simply being the automaton that does not contain $x$.

![Diagram](image)

Figure 2.14: No occurrences of $x$ DFA, $x = bab$
Algorithm 12 No occurrences of $x$ DFA

Let $x, y \in \Sigma^*$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w\} = \{w \mid x \notin w, w \in \Sigma^*\}$

Steps for DFA:

1. Create machine $M_1$ using Algorithm 3 with $w = x$
2. Take the complement of $M_1$

For completeness, when $\Sigma$ consists of a single character, the only solution for this problem lies in the edge case where we detect for no occurrences of $x$. No matter what $x$ and $y$ we choose, $x$ will always be the suffix of $xy$, so we are simply checking for strings with a length less than $x$.

2.4 The Consecutive Characters Problems

We will define a run as a sequence of some character $c$ appearing at least two times consecutively in a string. For example, $a$, is not a run, but $aa$, $aaa$, $aaaa$... are runs. In general, a run is defined by the regular expression $ccc^*$.

Before getting into the constructions we would like to note another important property that is present in the run constructions.

Property 3: States from $q_2$ to $q_n$ are associated with the length of the current run of $c$ detected at that point. For example, at state $q_3$, we currently have a run of size 3 detected.

2.4.1 Run of character $c$

Since a run is simply detecting a chain of two or more of some character $c$, this is the same problem as finding the substring of $cc$ in a string. If the run is longer than 2 characters, then we can still check for just $cc$, since $cc$ is always a substring of $ccc^*$. We have previously solved the substring problem with Algorithm 3, so this problem can be solved by using Algorithm 3 with $w = cc$. 25
2.4.2 All runs of character $c$ must be length $n$

To start our construction which checks if a string contains only runs of character $c$ of length $n$, we first need to create $n + 1$ states, labeled $q_0$ through $q_n$. We can now break the problem into 3 distinct sections:

- Detecting a run of $c$ ($q_0, q_1$)
- Checking that run is at least length $n$ ($q_2 - q_{n-1}$)
- Checking that run is no greater than length $n$ ($q_n$)

For detecting a run of $c$, it is identical to determining the transitions on “The Prefix States” from 2.2.1. We start by making $q_0$ the starting state. Then for all $i$ in the range of $[0, n - 1]$, we add the transition of $q_i$ to $q_{i+1}$ over $c$, with identical logic from Property 1. Since a run only begins after we detect two instances of our character $c$ consecutively, $q_0$ and $q_1$ can transition back to $q_0$ over $\Sigma - \{c\}$ since the run hasn’t started yet. For the same reasoning, we can make $q_0$ and $q_1$ final states, as we only have to worry about rejecting if are in the process of detecting a run. The transition on $c$ from $q_1$ to $q_2$ brings us to the next step of run detection, since at that point we will have detected a run of $c$ that is length 2.

As stated by Property 3, states $q_2$ through $q_n$ tell us how long of a run is currently detected by the number in the states subscript. For states $q_2$ through $q_{n-1}$, we do not need to add any more transitions, as transitions on any character other than $c$ will result in a run ending in one of these states, and because we only want runs of length $n$, we can immediately reject strings at this point.

The transition from $q_{n-1}$ to $q_n$ over $c$ gives us a run of exactly length $n$. If we get another $c$, then we know our current run of $c$’s has a length longer than $n$, and we should reject this string, since we have detected a run of $c$ that is not length $n$. For this reason, we do not add a transition
over \( c \) from \( q_n \). However if we get any other character in the alphabet, then we know we just finished a run of length \( n \), and can transition back to \( q_0 \) to begin detecting the next run. For this we add the transition from \( q_n \) to \( q_0 \) over \( \Sigma - c \). We also make \( q_n \) a final state, because if a string ends here, we ended with a run that was length \( n \).

The resulting construction is the NFA that describes our problem.

The resulting construction is the NFA that describes our problem.

![NFA Diagram](image)

**Figure 2.16: Runs of \( c \) must be length \( n \) NFA, \( c = a \), \( n = 4 \)**

**Algorithm 13** Runs of \( c \) must be length \( n \) NFA

Let \( c \in \Sigma \)

Let \( n \in \mathbb{Z}, \ n \geq 2 \)

Let \( \mathcal{L} = \{ w \mid w \in \Sigma^*, \ m \in \mathbb{Z}, \ m \geq 2, \ m \neq n, \ c^m \notin w \} \)

**Steps for NFA:**

1. Create \( n + 1 \) states, labeled \( q_0 \ldots q_n \)

2. Make \( q_0 \) the starting state

3. Make \( q_0, q_1, q_n \) the final states

4. For all \( i \) in \([0, n - 1]\), create the transitions \( q_i \xrightarrow{c} q_{i+1} \)

5. For \( i = 0 \) and \( i = 1 \), create the transition \( q_i \xrightarrow{\Sigma - \{c\}} q_0 \)

6. Add the transition \( q_n \xrightarrow{\Sigma - \{c\}} q_0 \)

To create our DFA for the above problem, we can start with our NFA solution as a base, and resolve all missing transitions to a dead state \( q_{rip} \). From \( q_n \), we add a transition to \( q_{rip} \) over \( c \), and from \( q_2 \) through \( q_{n-1} \) we add transitions to \( q_{rip} \) over \( \Sigma - c \), along with a self-loop over \( q_{rip} \) on \( \Sigma \).
Algorithm 14 All runs of $c$ must be length $n$ DFA

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $L = \{w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \neq n, c^m \notin w\}$

Steps for DFA:

1. Use the construction from Algorithm 13 with $c = c$, $n = n$

2. Create a state labeled $q_{rip}$

3. Add the transition $q_n \xrightarrow{c} q_{rip}$

4. For all $i$ in $[2, n-1]$, create the transitions $q_i \xrightarrow{\Sigma - \{c\}} q_{rip}$

5. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$

When $\Sigma$ is a single character, every string greater than length 1 is a run, and here we check for only the string with length $n$. The above constructions will still hold, and we simply accept all strings with length 0, 1 and $n$.

2.4.3 No runs of character $c$ are length $n$

If we want to check that there are no runs of $c$ that are length $n$, it is not as simple as taking the complement of the previous construction, as that would accept if there was at least one run of $c$ that was not length $n$. 
To start our NFA construction, we first create \( n + 2 \) states, labeled \( q_0 \) through \( q_{n+1} \). After making \( q_0 \) our starting state, we use a similar technique as demonstrated in Property 1, creating a chain of transitions between our states over \( c \), where for all \( i \) in \([0, n]\), we add a transition from \( q_i \) to \( q_{i+1} \) over \( c \). We then add a self-loop on \( q_{n+1} \) over \( c \).

From Property 3, the current state number represents the length of the current run we have detected (except for \( q_{n+1} \), which represents a run greater than \( n \)), when the state number is greater than 1. This means that if we get a character than isn’t \( c \) when in state \( q_n \), our run will have ended with a length of \( n \), and we should never accept. For this reason, we simply do not add a transition on \( q_n \) over \( \Sigma - \{c\} \). Following a similar chain of logic, if we end in state \( q_n \), we should also not accept, so we leave \( q_n \) as a non-accepting state. Every other state however should be an accepting state, since we know we do not have a run of length \( n \) in those states.

Finally for all states except \( q_n \), we should add a transition over \( \Sigma - \{c\} \) back to \( q_0 \), where we can start detecting the next run of \( c \). Formally, for all \( i \) in \([0, n + 1]\) except \( n \), add the transition from \( q_i \) to \( q_0 \) over \( \Sigma - \{c\} \), and make all states \( q_i \) accepting.

Figure 2.18: No runs of \( c \) with length \( n \) NFA, \( c = a, n = 4 \)
**Algorithm 15** All runs of character $c$ are not length $n$ NFA

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, c^n \notin w\}$

**Steps for NFA:**

1. Create $n + 2$ states, labeled $q_0 \ldots q_{n+1}$

2. Make $q_0$ the starting state

3. Make all states except $q_n$ final states

4. For all $i$ in $[0, n]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$

5. For all $i$ in $[0, n+1]$ except $n$, create the transitions $q_i \xrightarrow{\Sigma - \{c\}} q_0$

6. Add self-loop transition $q_{n+1} \xrightarrow{c} q_{n+1}$

The DFA construction of the above algorithm is simple, as all we need is a state $q_{rip}$ to our NFA where $q_n$ will transition to over $\Sigma - \{c\}$. Then we add a self-loop on $q_{rip}$ over $\Sigma$ and our DFA is complete.

![Diagram](https://via.placeholder.com/150)

**Figure 2.19**: No runs of $c$ with length $n$ DFA, $c = a$, $n = 4
Algorithm 16 All runs of character $c$ are not length $n$ DFA

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, c^n \notin w\}$

Steps for DFA:

1. Use the construction from Algorithm 15 with $c = c$ and $n = n$

2. Create a state labeled $q_{rip}$

3. Add the transition $q_n \xrightarrow{\Sigma \setminus \{c\}} q_{rip}$

4. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$

Similar to the last construction, when $\Sigma$ is a single character, every string greater than length 1 is a run, and since we are now checking that all runs are not length $n$, we are simply checking that the length of our string is not length 0, 1, or $n$. For this we can use the above NFA construction (Algorithm 15) for both the NFA and DFA of this case, as all $\Sigma \setminus \{c\}$ transitions will be the empty set, and will be omitted.

2.4.4 All runs of $c$ must be lengths in $M$

In 2.4.2, we discussed finding strings that only contain runs of some character $c$ that are a single length $n$. We expand upon that algorithm here, to allow strings to contain multiple different lengths of runs. For this, we define a set $M$ that contains all the lengths of runs we will accept in a given input string. All elements in $M$ must be integers, and 2 or greater.

To start our NFA construction, we first need to find the maximum element in $M$, which we will call $m$. From this maximum element $m$, we use Algorithm 13 with $n = m$ as a base. Since $m$ is our maximum run length, if we detect a run longer than $m$ we can reject the string, which our base automaton will handle. For elements less than $m$ however, we have the option of continuing our run detection, or stopping at that point and moving on to detecting the next run of $c$. In 2.4.2 we covered the three sections of our automaton with the third section consisting of just the state $q_{|m|}$ which ensures a run is not longer than length $m$. For our other elements in $M$, we should add
transitions to $q_0$ as well as make these states final, so they share the same accepting properties as $q_m$. To properly define the rest of $M$, we create a set $M'$ that is all elements in $M$ except for $m$. Then for each element $n$ in $M'$, we add the transition $q_n$ to $q_0$ over $\Sigma - c$. We also make all states $q_n$ final.

With this we have the NFA construction for the above problem.

![NFA Diagram](image)

**Figure 2.20:** Only runs of $c$ with lengths in $M$ NFA, $c = a$, $M = \{2, 4\}$

**Algorithm 17** All runs of $c$ must be lengths in $M$ NFA

Let $c \in \Sigma$

Let $M \subseteq \{n \mid n \in \mathbb{Z}, n \geq 2\}$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \notin M, c^m \notin w\}$

**Steps for NFA:**

1. Let $m$ be the maximum element in $M$

2. Use the construction from Algorithm 13 with $n = m$

3. Let $M' = M - \{m\}$

4. For all $n \in M'$, add the transition $q_n \xrightarrow{\Sigma - c} q_0$

5. For all $n \in M'$, make $q_n$ final

For our DFA of the above problem, we can use the NFA we just created as a base. Similar to many other constructions, we simply need to resolve all other transitions missing by adding a dead state $q_{rip}$. We start by adding the transition from $q_n$ to $q_{rip}$ over $c$. Then for all states that are not
final, we add the missing transitions to \( q_{rip} \). Formally, we can state that for all \( n \geq 2, n \notin M \), add the transition \( q_n \) to \( q_{rip} \) over \( \Sigma - c \). Finally, add the self-loop to \( q_{rip} \) over \( \Sigma \).

![Diagram](image)

Figure 2.21: Only runs of \( c \) with lengths in \( M \) DFA, \( c = a, M = \{2,4\} \)

---

**Algorithm 18** All runs of \( c \) must be lengths in \( M \) DFA

Let \( c \in \Sigma \)

Let \( M \subseteq \{ n \mid n \in \mathbb{Z}, n \geq 2 \} \)

Let \( \mathcal{L} = \{ w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \notin M, c^m \notin w \} \)

**Steps for DFA:**

1. Use the construction from Algorithm 17 with \( c = c \) and \( M = M \)

2. Let \( m \) be the maximum element in \( M \)

3. Let \( M' = M - \{m\} \)

4. For all \( n \in M' \), make \( q_n \) a final state

5. For all \( n \) in \([2, m - 1]\), if \( n \notin M' \), add the transition \( q_n \xrightarrow{\Sigma - \{c\}} q_{rip} \)

---

Finally we must once again discuss if \( \Sigma \) is a single character. In this case, we are detecting when the length of our string is one of the elements in \( M \cup \{0,1\} \). A simple construction would be using the Algorithm 8 NFA or Algorithm 9 DFA with \( w = c^m \), where \( m \) is the maximum element of \( M \), and then for all remaining elements in \( M \cup \{0,1\} \), \( n \), make \( q_n \) a final state.
2.5 The Conditional Counting Modulo n Problem

This algorithm generalizes a common problem students may encounter, where for example, they will need to design an automaton that only accepts when a string has an even number of $a$’s. This construction allows a lot of variability in the initial parameters set, as there are 4 parameters to choose from. First, some subset of the alphabet, $A$, which contains all the characters we would like to keep track of the sum of, that is, count the total number of all of these characters in a given input string. Then two positive integers $x$ and $y$, where $x$ is our divisor, and $y$ is our remainder, which must be less than or equal to $x$. Finally we can choose some relation in $C (\{=, <, >, \neq, \leq, \geq\})$ to compare the result of the division with our desired remainder, and only accept strings that fit those criteria.

The NFA and DFA are identical for this problem. To start, we create $x$ states labeled $q_0$ through $q_{x-1}$. Due to the looping nature of modulus, we only need $x$ states to keep track of the $x$ unique equivalence classes. After making $q_0$ our starting state, we add transitions between each state similar to Property 1, but instead of the transitions being a single character, they will be over $A$. We will also add self-loop transitions for every state on $\Sigma - A$, as characters outside of $A$ will not affect the count of characters we are checking for. Formally, for all $i$ in $[0, x - 2]$, add the transitions $q_i$ to $q_{i+1}$ over $A$, and self-loop on $q_i$ over $\Sigma - A$. We then have $q_{x-1}$ transition back to $q_0$ over $A$, and also add the self-loop on $q_{x-1}$ over $\Sigma - A$.

The final step is to determine which states should be our accepting states, which are dependant on which condition in $C$ is chosen.

(=) The simplest case is when we want our result equal to $y$, because then the only state where the result would be true is in state $q_y$, which we would make final.

(<) If we only want to accept when our result is less than $y$, then we would want all the states before $q_y$ to be accepting, which would be $q_0$ through $q_{y-1}$.

(>) If we only want to accept when our result is greater than $y$, then we would want all the states after $q_y$ to be accepting, which would be $q_{y+1}$ through $q_{x-1}$.

(\neq) If we only want to accept when our result is not equal than $y$, then we would want all the states except $q_y$ to be accepting, which would be $q_0$ through $q_{y-1}$ and $q_{y+1}$ through $q_{x-1}$.
If we only want to accept when our result is less than or equal to $y$, then we would want all the states before and including $q_y$ to be accepting, which would be $q_0$ through $q_y$.

If we only want to accept when our result is greater than or equal to $y$, then we would want all the states after and including $q_y$ to be accepting, which would be $q_y$ through $q_{x-1}$.

Figure 2.22: Modulo Construction, $x = 4$, $y = 2$, $A = \{a\}$, $c = (=)$
**Algorithm 19** Modulo Construction

Let $w \in \Sigma^*$

Let $x, y \in \mathbb{N}$, $x > y$

Let $A \subseteq \Sigma$

Let $|w_A|$ be the count of the characters contained in $A$ that are found in $w$

Let $C = \{=, <, >, \neq, \leq, \geq\}$

Let $\mathcal{L} = \{w \mid |w_A| \% x \ c \ y, \ c \in C\}$

**Steps for NFA and DFA:**

1. Create $x$ states, labeled $q_0 \ldots q_{x-1}$

2. Make $q_0$ the starting state

3. For all $i$ in $[0, x-2]$, add transitions $q_i \xrightarrow{A} q_{i+1}$ and self-loop transitions $q_i \xrightarrow{\Sigma-A} q_{i+1}$

4. Create the transition $q_{x-1} \xrightarrow{A} q_0$ and self-loop transition $q_{x-1} \xrightarrow{\Sigma-A} q_{x-1}$

5. Based on $c$, make the following states final:
   
   $=$ - $q_y$
   $<$ - $q_0 \ldots q_{y-1}$
   $>$ - $q_{y+1} \ldots q_{x-1}$
   $\neq$ - $q_0 \ldots q_{y-1}, q_{y+1} \ldots q_{x-1}$
   $\leq$ - $q_0 \ldots q_y$
   $\geq$ - $q_y \ldots q_{x-1}$

When $\Sigma$ is a single character, the above construction still holds, and all the self-loops simply don’t exist anymore, since $\Sigma - A$ will always be the empty set. An alternate interpretation of the above problem now depends on the length of $w$ instead of the count of certain characters in $w$, due to all characters being the same.
2.6 The Periodic Character Detection Problem

This construction checks that one or more characters are always present at a specific repeating cycle in our string. For example, if one wanted to check that there is always an \( a \) at every third position with an offset of 1, then a string \( w = babbabaah \) would satisfy that condition. The three parameters specified here are similar to the ones in the previous Modulo construction, where we have a set of characters \( A \) which is a subset of \( \Sigma \), and two integers \( x \) and \( y \), that must be non-negative, with \( x \) being greater than \( y \). \( x \) represents our divisor, and \( y \) is the result we want to check for. Also note that this algorithm 0-indexes our strings.

We start in a similar fashion to the Modulo construction, by creating \( x \) states labeled \( q_0 \) through \( q_{x-1} \), and making \( q_0 \) the starting state. For our NFA construction we simply make all states final, and add transitions between every consecutive state over \( \Sigma \), except for \( q_y \), and have \( q_{x-1} \) loop back to \( q_0 \). Formally, for all \( i \) in \([0, x - 1]\) except \( y \), add the transition from \( q_i \) to \( q_{(i+1) \% x} \) over \( \Sigma \). To ensure we only get characters in \( A \) at the offset \( y \), we add the transition from \( q_y \) to \( q_{(y+1) \% x} \) over \( A \). Note the use of modulus for indexing the “to” state, to ensure we loop back to \( q_0 \) when \( i \) or \( y \) is \( x - 1 \).

![Regularly repeating characters NFA](image)

Figure 2.23: Regularly repeating characters NFA, \( x = 4 \), \( y = 2 \), \( A = \{a\} \)
Algorithm 20 Specific characters at every $x^{th}$ position with offset $y$ NFA

Let $A \subseteq \Sigma$

Let $x, y \in \mathbb{N} \cup \{0\}$, $x > y$

Let $\mathcal{L} = \{ w \in \Sigma^*, \forall \text{index } \in \{ m | 0 \leq m \leq |w| - 1, m \% x = y \}, w_{\text{index}} \in A \}$

(In plain text, every $x^{th}$ character plus an offset of $y$ of $w$ must be a character from the set $A$.)

Steps for NFA:

1. Create $x$ states, labeled $q_0 \ldots q_{x-1}$
2. Make $q_0$ the starting state
3. Make all states final
4. For all $i$ in $[0, x - 1]$, except $y$, add transitions $q_i \xrightarrow{\Sigma} q_{(i+1) \% x}$
5. Add a transition $q_y \xrightarrow{A} q_{((y+1) \% x)}$

Similar to many other constructions, the DFA simply adds a dead state and resolves all missing transitions to the dead state. In our case with the previous construction as a base, we add a state $q_{\text{rip}}$ where $q_y$ transitions to $q_{\text{rip}}$ over $\Sigma - A$, which is all other characters in our alphabet.

![Figure 2.24: Regularly repeating characters DFA, $x = 4$, $y = 2$, $A = \{a\}$](image)
Algorithm 21 Specific Characters at every $x^{th}$ position with offset $y$ DFA

Let $A \subseteq \Sigma$

Let $x, y \in \mathbb{N} \cup \{0\}$, $x > y$

Let $\mathcal{L} = \{ w \mid w \in \Sigma^*, \forall \text{index} \in \{ m \mid 0 \leq m \leq |w| - 1, \ m \% x = y \}, \ w[\text{index}] \in A \}$

(In plain text, every $x^{th}$ character plus an offset of $y$ in $w$ must be a character from the set $A$.)

Steps for DFA:

1. Use the construction from Algorithm 20 with $x = x$, $y = y$, $A = A$
2. Create state $q_{rip}$
3. Add a transition $q_y \xrightarrow{\Sigma - y} q_{rip}$
4. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$

If $\Sigma$ is a single character, then $A$ must equal $\Sigma$, and since every state is accepting, we end up with a machine that accepts every possible string, which can be reduced to a single accepting state with a self-loop.

2.7 The Contains X But Not Y Problem

This problem looks to construct the automaton that only accepts a string $w$ if some string $x$ is a substring of $w$, but another string $y$ is not a substring of $w$.

If $y$ is a substring of $x$, then the language is the empty set. This is due to the fact that we can never get an instance of $x$ without also getting an instance of $y$. The automaton for this is simply a single non-accepting state that has a self-loop over $\Sigma$.

If $x$ is a prefix of $y$, we can drastically reduce the runtime and number of states with the following algorithm. By default, this problem could be solved by taking the intersection of $x$ with the complement of $y$. Using the naive intersection method, we would end up with $(|x| + 1) \ast (|y| + 1)$ states in our DFA. The below algorithm produces a DFA with fewer states, $(|x| + 1) + (|y| + 1)$. 

39
This reduction in states comes from not having to search for both \( x \) and \( y \) at the same time, since we know \( y \) will only come after we detect \( x \), due to \( x \) being a prefix of \( y \).

The construction of this DFA heavily relies on other constructions and algorithms mentioned throughout this paper. We start with creating two automata, \( M_1 \) and \( M_2 \), where \( M_1 \) is the automaton produced by Algorithm 3 with \( w = x \), and \( M_2 \) is the automaton produced by Algorithm 3 with \( w = y \). \( M_1 \) allows us to detect when we get \( x \) as a substring in \( w \), and \( M_2 \) allows us to detect an occurrence of \( y \). Since we want to detect when we don’t get an occurrence of \( y \), we create a new machine \( M_3 \) which is just the complement of \( M_2 \). For clarity, we also relabel all the states in \( M_3 \) from \( q_i \) to \( r_i \) for all \( i \) in \([0, |y|]\).

Now that we have our two base automata, we need to connect them together somehow to allow us to find an occurrence of \( x \) without \( y \). We do this in a very similar fashion to steps 3 and 4 in Algorithm 7, the Suffix DFA. We first remove the initial state on \( r_0 \), since we are going to connect the two automaton, and \( q_0 \) will remain the start. Then we remove the self-loop on \( q_{|x|} \), as we will be adding new transitions of every character in \( \Sigma \) to connect to \( M_3 \). Now to add those new transitions, for every character \( c \) in \( \Sigma \), we use Algorithm 1 with \( p = y \) and \( s = xc \) to find the overlap between the prefix of \( y \) and the suffix of \( xc \), calling the resulting string \( m_c \). These strings tell us how much of \( y \) will be detected after we find our first occurrence of \( x \), plus whatever the next character to transition on is. We can then add the transitions \( q_{|x|} \) to \( r_{|m_c|} \) over \( c \) for all \( c \) in \( \Sigma \).

Figure 2.25: Contains \( x \) but not \( y \) DFA, \( x = aaba \), \( y = aababa \)
Algorithm 22 Contains $x$ but not $y$

Let $x, y \in \Sigma^*$

Let $\mathcal{L} = \{ w \mid w \in \Sigma^*, x \in w, y \notin w, x \in p \text{ or } x \neq y \}$

Steps for DFA:

1. Create machine $M_1$ using Algorithm 3 with $w = x$
2. Create machine $M_2$ using Algorithm 3 with $w = y$
3. Let $M_3$ be the complement of $M_2$ \(^1\)
4. Relabel the states in $M_3$ from $q_0$ through $q_{|y|}$ to $r_0$ through $r_{|y|}$
5. Make $r_0$ no longer an initial state
6. Remove self-loop on $q_{|x|}$
7. For all $c \in \Sigma$, let $m_c$ be the output from Algorithm 1 with $p = y$ and $s = xc$
8. Create the transition $q_{|x|} \xrightarrow{c} r_{|m_c|}$ for all $c \in \Sigma$

When $\Sigma$ is a single character, the problem boils down to all strings $w$ that are at least length $x$, but shorter than length $y$, $x \leq |w| < y$. The above DFA construction can be used, and to remove redundant states, remove all states $r_i$ and their associated transitions where $i \leq x$.

\(^1\)The complement of a DFA can be computed in polynomial time by swapping the final and non-final states. [16]
CHAPTER 3
Implementation

In this chapter, we discuss the software implementation of some of the above constructions. The code is broken up into 4 main sections, which will be gone through in more depth throughout the chapter:

- Main
- Automaton Classes
- Helper
- Construction Classes

The software presented allows users to select one of the above constructions, and after filling in the parameters specified for the given construction, will create an XML file that is readable by JFLAP, along with opening up the generated automaton in JFLAP. Note that at the time of completing this thesis, only the substring problems have been implemented in the software, with the other constructions to be added soon.

It should be noted that this software is not directly connected with JFLAP in any way, but uses their XML format when generating automata, making the outputs from our program compatible with JFLAP. None of the code modifies JFLAP itself in any way. For convenience during development however, when an automaton is generated with our program it automatically opens the automaton in JFLAP via the terminal.

3.1 Main

The “Main” section is comprised of Main.java alone. This section is where the program is ran from, and contains all the code to facilitate a user interacting with the software. The user interacts with the program through the command line, with basic error handling in place to ensure the user gets an expected output.
3.2 Automaton Classes

The “Automaton Classes” consist of 3 files: State.java, Transition.java, and Automaton.java. These classes allow us to construct our automaton in a simple and modular fashion, allowing easy additions for more construction algorithms. These classes also hold the code to produce the JFLAP readable XML code.

3.2.1 State.java

State.java keeps track of all information about the states of our automaton. The following attributes are tied to each state:

- **id** (int): A unique identifier for each state, which is needed by JFLAP
- **name** (String): An optional entry that allows states in JFLAP to have a custom display name, such as $q_rip$, instead of a default name
- **initial** (boolean): Marks the state as the starting state of the automaton
- **accepting** (boolean): Marks the state as a final state of the automaton
- **x** (double): Holds the x coordinate of this state, for JFLAP to draw the image
- **y** (double): Holds the y coordinate of this state, for JFLAP to draw the image

An interesting feature of having to keep track of the $x$ and $y$ coordinates is that our construction implementations can create and place the states in a rigorous fashion based on our interpretation of the best way to present the given automaton.

3.2.2 Transition.java

Transition.java stores all of the transitions present in our automaton. The attributes of a transition are as follows:

- **from** (State): The state this transition originates from
• **to** (State): The state this transition leads into

• **label** (String): The character this transition transitions over

JFLAP does allow for advanced positioning of transitions, however, in this first iteration of the software, we only have basic transitions drawn to JFLAP. There are future plans to allow for more complex transition arrows to be drawn.

### 3.2.3 Automaton.java

Automaton.java simply holds two lists, one for States, **states**, and one for Transitions **transitions**. The main job of this class is to generate the JFLAP XML for drawing the automaton, which builds upon the XML sections the State and Transition classes handle.

### 3.3 Helper

The “Helper” section contains a single class, Helper.java. This file provides the constructions with useful intermediary steps needed during construction. The two methods in Helper are:

- **String overlap(String prefix, String suffix)**: Returns the output as defined by Algorithm 1, with \( p = \text{prefix} \) and \( s = \text{suffix} \)

- **String longestPrefix(String w)**: Returns the longest prefix of a single character, as needed in Algorithm 7 Step 5.

Separating these methods from the constructions themselves are useful, even if the code is only used in a single construction, because it allows the construction code to closely follow the steps defined in this thesis.

### 3.4 Construction Classes

Currently, the “Construction Classes” only consist of Substring.java, which covers the constructions in section 2.2.1 through 2.2.3. The methods and constructions currently covered include:
• Automaton `substringnfa(String w, char[] alphabet)`: Construction defined by Algorithm 2

• Automaton `substringdfa(String w, char[] alphabet)`: Construction defined by Algorithm 3

• Automaton `prefixnfa(String w, char[] alphabet)`: Construction defined by Algorithm 4

• Automaton `prefixedfa(String w, char[] alphabet)`: Construction defined by Algorithm 5

• Automaton `suffixnfa(String w, char[] alphabet)`: Construction defined by Algorithm 6

• Automaton `suffixdfa(String w, char[] alphabet)`: Construction defined by Algorithm 7

Work has been started on algorithms covered by section 2.3, however the Automaton class needs to be expanded to easily remove States and Transitions before it can be completed.
CHAPTER 4
Conclusion and Future Work

In this thesis, we introduced generalized NFA constructions and DFA constructions for the following languages:

- All strings that contain a substring \( w \)
- All strings that start with a prefix \( w \)
- All strings that end with a suffix \( w \)
- All strings that are \( w \) (Which is just \( w \))
- All strings \( x \) must be immediately followed by string \( y \)
- All strings that contain a run of some character \( c \)
- All strings where all runs of \( c \) must be length \( n \)
- All strings where no runs of \( c \) are length \( n \)
- All strings where runs of \( c \) must be a length defined in set \( M \)
- All strings that follow modulo properties of character counts
- All strings that have characters defined in set \( A \) at regularly repeating intervals
- All strings that contain some string \( x \) but string \( y \) when \( x \) is a prefix of \( y \) (No NFA given)

We also introduced and solved the maximum prefix-suffix overlap problem, which is a crucial intermediary step to some of the constructions above.

Finally, we implemented some of these algorithms in software, to produce XML code compatible with JFLAP to visualize the automata generated from the constructions in order to further understand the patterns and structures that arise from these constructions.
An immediate next step is to implement all of the constructions presented in this thesis in software. This will help to further visualize the algorithms presented. The software would be an educational tool similar to JFLAP [14], with the known algorithms such as combining automata with known closures, converting from NFA to DFA, and minimizing DFAs as well as the algorithms presented in this thesis.

Another open area is to continue to generalize automaton constructions for the regular languages. Another open area related to this research is to look at developing algorithms for context-free languages.

Lastly, developing a system that can take in a language described in set notation and produce an automaton for the language would be a great tool for educators and students.
BIBLIOGRAPHY


APPENDIX A

Constructions with No References

Below are all constructions in the order they were presented in the paper, separated by section, with all instructions written out, instead of relying on calling other constructions to use as a base. Constructions that have been modified by resolving the reference will be marked with a “*” in their title. Note that Algorithm 1 will still be referenced as it is not a construction, along with any well known closures that a construction may use.

A.1 The Substring Problems

A.1.1 Substring Constructions

Algorithm 23 Substring NFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{ xwy \mid x, y \in \Sigma^*, x, y \neq \lambda \} \)

Steps for NFA:

1. Create \(|w| + 1\) states, labeled \(q_0 \ldots q_{|w|}\)

2. Make \(q_0\) the starting state

3. Make \(q_{|w|}\) the final state

4. For all \( i \) in \([0, |w| - 1]\), create the transition \(q_i \xrightarrow{w_i} q_{i+1}\)

5. Add self-loop transition \(q_0 \xrightarrow{\Sigma} q_0\)

6. Add self-loop transition \(q_{|w|} \xrightarrow{\Sigma} q_{|w|}\)
Algorithm 24 Substring DFA

Let $w \in \Sigma^*$

Let $L = \{xwy \mid x, y \in \Sigma^*, x, y \neq \lambda\}$

Steps for DFA:

1. Create $|w| + 1$ states, labeled $q_0 \ldots q_{|w|}$

2. Make $q_0$ the starting state

3. Make $q_{|w|}$ the final state

4. For all $i$ in $[0, |w| - 1]$, create the transitions $q_i \xrightarrow{w_i} q_{i+1}$

5. Add self-loop transition $q_{|w|} \xrightarrow{\Sigma} q_{|w|}$

6. Let $p$ be the longest prefix of $w$ consisting only of the character $w_0$

7. Add self-loop transition $q_{|p|} \xrightarrow{w_0} q_{|p|}$

8. For all $i$ in $[|p| + 1, |w| - 1]$ add transitions $q_i \xrightarrow{\{w_0\} - \{w_i\}} q_{1}$

9. For all $i$ in $[0, |w| - 1]$ add transitions $q_i \xrightarrow{\Sigma - \{w_0, w_i\}} q_{0}$
A.1.2 Prefix Constructions

Algorithm 25 Prefix NFA
Let $w \in \Sigma^*$
Let $\mathcal{L} = \{wy \mid y \in \Sigma^*, y \neq \lambda\}$

Steps for NFA:

1. Create $|w| + 1$ states, labeled $q_0 \ldots q_{|w|}$
2. Make $q_0$ the starting state
3. Make $q_{|w|}$ the final state
4. For all $i$ in $[0, |w| - 1]$, create the transition $q_i \xrightarrow{w_i} q_{i+1}$
5. Add self-loop transition $q_{|w|} \xrightarrow{\Sigma} q_{|w|}$

Algorithm 26 Prefix DFA*
Let $w \in \Sigma^*$
Let $\mathcal{L} = \{wy \mid y \in \Sigma^*, y \neq \lambda\}$

Steps for DFA:

1. Create $|w| + 1$ states, labeled $q_0 \ldots q_{|w|}$
2. Make $q_0$ the starting state
3. Make $q_{|w|}$ the final state
4. For all $i$ in $[0, |w| - 1]$, create the transition $q_i \xrightarrow{w_i} q_{i+1}$
5. Add self-loop transition $q_{|w|} \xrightarrow{\Sigma} q_{|w|}$
6. Create a state $q_{rip}$
7. For all $i$ in $[0, |w| - 1]$, create the transition $q_i \xrightarrow{\Sigma - \{w_i\}} q_{rip}$
8. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$
A.1.3 Suffix Constructions

**Algorithm 27** Suffix NFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{ xw \mid x \in \Sigma^*, \ x \neq \lambda \} \)

**Steps for NFA:**

1. Create \(|w| + 1\) states, labeled \(q_0 \ldots q_{|w|}\)

2. Make \(q_0\) the starting state

3. Make \(q_{|w|}\) the final state

4. For all \(i\) in \([0, |w| - 1]\), create the transition \(q_i \xrightarrow{w_i} q_{i+1}\)

5. Add self-loop transition \(q_0 \xrightarrow{\Sigma} q_0\)
Algorithm 28 Suffix DFA*

Let $w \in \Sigma^*$

Let $\mathcal{L} = \{xw \mid x \in \Sigma^*, x \neq \lambda\}$

Steps for DFA:

1. Create $|w| + 1$ states, labeled $q_0 \ldots q_{|w|}$

2. Make $q_0$ the starting state

3. Make $q_{|w|}$ the final state

4. For all $i$ in $[0, |w| - 1]$, create the transitions $q_i \xrightarrow{w_i} q_{i+1}$

5. Let $p$ be the longest prefix of $w$ consisting only of the character $w_0$

6. Add self-loop transition $q_{|p|} \xrightarrow{w_0} q_{|p|}$

7. For all $i$ in $[0, |w| - 1]$ except $|p|$, add transitions $q_i \xrightarrow{\{w_0\} - \{w_i\}} q_1$ and $q_i \xrightarrow{\Sigma - \{w_0, w_i\}} q_0$

8. Add the transition $q_{|p|} \xrightarrow{\Sigma - \{w_0, w_i\}} q_0$

9. For all $c \in \Sigma$, let $m_c$ be the output from Algorithm 1 where the prefix $p = w$ and the suffix $s = wc$

10. Create the transition $q_{|w|} \xrightarrow{c} q_{|m_c|}$ for all $c \in \Sigma$
A.1.4 Trivial Constructions

**Algorithm 29** Trivial NFA

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{w\} \)

**Steps for NFA:**

1. Create \( |w| + 1 \) states, labeled \( q_0 \ldots q_{|w|} \)
2. Make \( q_0 \) the starting state
3. Make \( q_{|w|} \) the final state
4. For all \( i \in [0, |w| - 1] \), create the transition \( q_i \xrightarrow{w_i} q_{i+1} \)

---

**Algorithm 30** Trivial DFA*

Let \( w \in \Sigma^* \)

Let \( \mathcal{L} = \{w\} \)

**Steps for DFA:**

1. Create \( |w| + 1 \) states, labeled \( q_0 \ldots q_{|w|} \)
2. Make \( q_0 \) the starting state
3. Make \( q_{|w|} \) the final state
4. For all \( i \in [0, |w| - 1] \), create the transition \( q_i \xrightarrow{w_i} q_{i+1} \)
5. Create a state \( q_{rip} \)
6. For all \( i \in [0, |w| - 1] \) create the transition \( q_i \xrightarrow{\Sigma - \{w_i\}} q_{rip} \)
7. Add self-loop transition \( q_{rip} \xrightarrow{\Sigma} q_{rip} \)
8. Create the transition \( q_{|w|} \xrightarrow{\Sigma} q_{rip} \)
A.2  The X Followed By Y Problem

Algorithm 31 $x$ followed by $y$ NFA

Let $x, y \in \Sigma^* \\

Let $L = \{w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w\}$

Steps for NFA:

1. Let $x'$ be $x_0 \ldots x_{|x| - 2}$

2. Create $|x'| + 1$ states, labeled $q_0 \ldots q_{|x'|}$

3. Make $q_0$ the starting state

4. Make $q_{|x'|}$ the final state

5. For all $i$ in $[0, |x'|-1]$, create the transitions $q_i \xrightarrow{x'_0} q_{i+1}$

6. Let $p$ be the longest prefix of $x'$ consisting only of the character $x'_0$.

7. Add self-loop transition $q_{|p|} \xrightarrow{x'_0} q_{|p|}$

8. For all $i$ in $[|p| + 1, |x'|-1]$ add transitions $q_i \xrightarrow{{x'_0}-{x'_i}} q_1$

9. For all $i$ in $[0, |x'|-1]$ add transitions $q_i \xrightarrow{\Sigma-{x'_0}-{x'_i}} q_0$

10. Make states $q_0 \ldots q_{|x'|}$ final

11. Add $|y|$ states labeled $r_0 \ldots r_{|y|-1}$

12. Add a transition from $q_{|x|-1} \xrightarrow{x_{|x|-1}} r_0$

13. Add the transitions $q_{|x|-1} \xrightarrow{\{x_0\}-{x_{|x|-1}}} q_1$ and $q_{|x|-1} \xrightarrow{\Sigma-{x_0}-{x_{|x|-1}}} q_0$

14. For all $i$ in $[0, |y|-2]$, add transitions $r_i \xrightarrow{y_i} r_{i+1}$.

15. Let $m$ be the output from Algorithm 1 where the prefix $p = x$ and the suffix $s = y$

16. Create a transition $r_{|y|-1} \xrightarrow{y_{|y|-1}} q_m$
Algorithm 32 No occurrences of $x$ DFA

Let $x, y \in \Sigma^*$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, \#(x)_w = \#(xy)_w\} = \{w \mid x \notin w, w \in \Sigma^\ast\}$

Steps for DFA:

1. Create $|x| + 1$ states, labeled $q_0 \ldots q_{|x|}$
2. Make $q_0$ the starting state
3. Make all states except $q_{|x|}$ final states
4. For all $i$ in $[0, |x| - 1]$, create the transitions $q_i \xrightarrow{x_i} q_{i+1}$
5. Add self-loop transition $q_{|x|} \xrightarrow{\Sigma} q_{|x|}$
6. Let $p$ be the longest prefix of $x$ consisting only of the character $x_0$
7. Add self-loop transition $q_{|p|} \xrightarrow{x_0} q_{|p|}$
8. For all $i$ in $[|p| + 1, |x| - 1]$ add transitions $q_i \xrightarrow{\{x_0\} - \{x_i\}} q_1$
9. For all $i$ in $[0, |x| - 1]$ add transitions $q_i \xrightarrow{\Sigma -\{x_0, x_i\}} q_0$
A.3 The Consecutive Characters Problems

A.3.1 All runs of character $c$ must be length $n$

Algorithm 33 Runs of $c$ must be length $n$ NFA

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $L = \{w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \neq n, c^m \notin w\}$

Steps for NFA:

1. Create $n + 1$ states, labeled $q_0 \ldots q_n$

2. Make $q_0$ the starting state

3. Make $q_0$, $q_1$, $q_n$ the final states

4. For all $i$ in $[0, n - 1]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$

5. For $i = 0$ and $i = 1$, create the transition $q_i \xrightarrow{\Sigma \setminus \{c\}} q_0$

6. Add the transition $q_n \xrightarrow{\Sigma \setminus \{c\}} q_0$
Algorithm 34 All runs of $c$ must be length $n$ DFA*
Let $c \in \Sigma$
Let $n \in \mathbb{Z}$, $n \geq 2$
Let $\mathcal{L} = \{w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \neq n, c^m \not\in w\}$

Steps for DFA:

1. Create $n + 1$ states, labeled $q_0 \ldots q_n$
2. Make $q_0$ the starting state
3. Make $q_0$, $q_1$, $q_n$ the final states
4. For all $i$ in $[0, n - 1]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$
5. For $i = 0$ and $i = 1$, create the transition $q_i \xrightarrow{\Sigma \setminus \{c\}} q_0$
6. Add the transition $q_n \xrightarrow{\Sigma \setminus \{c\}} q_0$
7. Create a state labeled $q_{rip}$
8. Add the transition $q_n \xrightarrow{c} q_{rip}$
9. For all $i$ in $[2, n - 1]$, create the transitions $q_i \xrightarrow{\Sigma \setminus \{c\}} q_{rip}$
10. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$
A.3.2 No runs of character $c$ are length $n$

**Algorithm 35** All runs of character $c$ are not length $n$ NFA

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, c^n \notin w\}$

Steps for NFA:

1. Create $n + 2$ states, labeled $q_0 \ldots q_{n+1}$
2. Make $q_0$ the starting state
3. Make all states except $q_n$ final states
4. For all $i$ in $[0, n]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$
5. For all $i$ in $[0, n+1]$ except $n$, create the transitions $q_i \xleftarrow{\Sigma - \{c\}} q_0$
6. Add self-loop transition $q_{n+1} \xrightarrow{c} q_{n+1}$
Algorithm 36 All runs of character $c$ are not length $n$ DFA* 

Let $c \in \Sigma$

Let $n \in \mathbb{Z}$, $n \geq 2$

Let $\mathcal{L} = \{ w \mid w \in \Sigma^*, c^n \notin w \}$

Steps for DFA:

1. Create $n + 2$ states, labeled $q_0 \ldots q_{n+1}$

2. Make $q_0$ the starting state

3. Make all states except $q_n$ final states

4. For all $i$ in $[0, n]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$

5. For all $i$ in $[0, n+1]$ except $n$, create the transitions $q_i \xrightarrow{\Sigma \setminus \{c\}} q_0$

6. Add self-loop transition $q_{n+1} \xrightarrow{c} q_{n+1}$

7. Create a state labeled $q_{rip}$

8. Add the transition $q_n \xrightarrow{\Sigma \setminus \{c\}} q_{rip}$

9. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$
A.3.3 All runs of $c$ must be lengths in $M$

**Algorithm 37 All runs of $c$ must be lengths in $M$ NFA**

Let $c \in \Sigma$

Let $M \subseteq \{n \mid n \in \mathbb{Z}, n \geq 2\}$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \notin M, c^m \notin w\}$

**Steps for NFA:**

1. Let $m$ be the maximum element in $M$

2. Create $m + 1$ states, labeled $q_0 \ldots q_m$

3. Make $q_0$ the starting state

4. Make $q_0$, $q_1$ the final states

5. For all $i$ in $[0, m - 1]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$

6. For $i = 0$ and $i = 1$, create the transition $q_i \xrightarrow{\Sigma - \{c\}} q_0$

7. For all $n \in M$, add the transition $q_n \xrightarrow{\Sigma - c} q_0$

8. For all $n \in M$, make $q_n$ final
**Algorithm 38** All runs of $c$ must be lengths in $M$ DFA

Let $c \in \Sigma$

Let $M \subseteq \{ n \mid n \in \mathbb{Z}, n \geq 2 \}$

Let $\mathcal{L} = \{ w \mid w \in \Sigma^*, m \in \mathbb{Z}, m \geq 2, m \notin M, c^m \notin w \}$

**Steps for DFA:**

1. Let $m$ be the maximum element in $M$

2. Create $m + 1$ states, labeled $q_0 \ldots q_m$

3. Make $q_0$ the starting state

4. Make $q_0, q_1$ the final states

5. For all $i$ in $[0, m - 1]$, create the transitions $q_i \xrightarrow{c} q_{i+1}$

6. For $i = 0$ and $i = 1$, create the transition $q_i \xrightarrow{\Sigma - \{c\}} q_0$

7. Create a state labeled $q_{rip}$

8. Add the transition $q_m \xrightarrow{c} q_{rip}$

9. For all $n \in M$, make $q_n$ a final state

10. For all $n$ in $[2, m]$, if $n \in M$, add the transition $q_n \xrightarrow{\Sigma - \{c\}} q_0$

11. For all $n$ in $[2, m]$, if $n \notin M$, add the transition $q_n \xrightarrow{\Sigma - \{c\}} q_{rip}$

12. Add self-loop transition $q_{rip} \xrightarrow{\Sigma} q_{rip}$
A.4 The Conditional Counting Modulo n Problem

**Algorithm 39 Modulo Construction**

Let \( w \in \Sigma^* \)

Let \( x, y \in \mathbb{N} \), \( x > y \)

Let \( A \subseteq \Sigma \)

Let \( |w_A| \) be the count of the characters contained in \( A \) that are found in \( w \)

Let \( C = \{=, <, >, \neq, \leq, \geq\} \)

Let \( \mathcal{L} = \{w \mid |w_A| \% x c y, c \in C\} \)

**Steps for NFA and DFA:**

1. Create \( x \) states, labeled \( q_0 \ldots q_{x-1} \)

2. Make \( q_0 \) the starting state

3. For all \( i \) in \([0, x-2]\), add transitions \( q_i \xrightarrow{A} q_{i+1} \) and self-loop transitions \( q_i \xrightarrow{\Sigma-A} q_{i+1} \)

4. Create the transition \( q_{x-1} \xrightarrow{A} q_0 \) and self-loop transition \( q_{x-1} \xrightarrow{\Sigma-A} q_{x-1} \)

5. Based on \( c \), make the following states final:
   - \( = - q_y \)
   - \( < - q_0 \ldots q_{y-1} \)
   - \( > - q_{y+1} \ldots q_{x-1} \)
   - \( \neq - q_0 \ldots q_{y-1}, q_{y+1} \ldots q_{x-1} \)
   - \( \leq - q_0 \ldots q_y \)
   - \( \geq - q_y \ldots q_{x-1} \)
A.5 The Periodic Character Detection Problem

Algorithm 40 Specific characters at every $x^{th}$ position with offset $y$ NFA

Let $A \subseteq \Sigma$

Let $x, y \in \mathbb{N} \cup \{0\}$, $x > y$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, \forall \text{index} \in \{m \mid 0 \leq m \leq |w| - 1, m \% x = y\}, w_{\text{index}} \in A\}$

(In plain text, every $x^{th}$ character plus an offset of $y$ of $w$ must be a character from the set $A$.)

Steps for NFA:

1. Create $x$ states, labeled $q_0 \ldots q_{x-1}$
2. Make $q_0$ the starting state
3. Make all states final
4. For all $i$ in $[0, x - 1]$, except $y$, add transitions $q_i \xrightarrow{\Sigma} q_{((i+1) \% x)}$
5. Add a transition $q_y \xrightarrow{A} q_{((y+1) \% x)}$
Algorithm 41 Specific Characters at every $x^{th}$ position with offset $y$ DFA*

Let $A \subseteq \Sigma$

Let $x, y \in \mathbb{N} \cup \{0\}, x > y$

Let $\mathcal{L} = \{w \mid w \in \Sigma^*, \forall \text{index } \in \{m \mid 0 \leq m \leq |w| - 1, m \% x = y\}, w[\text{index}] \in A\}$

(In plain text, every $x^{th}$ character plus an offset of $y$ in $w$ must be a character from the set $A$.)

Steps for DFA:

1. Create $x$ states, labeled $q_0 \ldots q_{x-1}$

2. Make $q_0$ the starting state

3. Make all states final

4. For all $i$ in $[0, x-1]$, except $y$, add transitions $q_i \xrightarrow{\Sigma} q_{((i+1) \% x)}$

5. Add a transition $q_y \xrightarrow{A} q_{((y+1) \% x)}$

6. Create state $q_{rip}$

7. Add a transition $q_y \xrightarrow{\Sigma - y} q_{rip}$

8. Add self-loop transition $q_{rip} \xrightarrow{} q_{rip}$
The Contains X But Not Y Problem*

Algorithm 42 Contains $x$ but not $y$

Let $x, y \in \Sigma^*$

Let $L = \{ w \mid w \in \Sigma^*, x \in w, y \notin w, x \in_p y, x \neq y \}$

Steps for DFA:

1. Create $|x| + 1$ states, labeled $q_0 \ldots q_{|x|}$
2. Make $q_0$ the starting state
3. Make $q_{|x|}$ the final state
4. For all $i$ in $[0, |x| - 1]$, create the transitions $q_i \xrightarrow{x_i} q_{i+1}$
5. Let $p$ be the longest prefix of $x$ consisting only of the character $x_0$
6. Add self-loop transition $q_{|p|} \xrightarrow{x_0} q_{|p|}$
7. For all $i$ in $[|p| + 1, |x| - 1]$ add transitions $q_i \xrightarrow{\{x_0\} - \{x_i\}} q_1$
8. For all $i$ in $[0, |x| - 1]$ add transitions $q_i \xrightarrow{\Sigma - \{x_0, x_i\}} q_0$
9. Create $|y| + 1$ states, labeled $r_0 \ldots r_{|y|}$
10. Make all states except $r_{|y|}$ final states
11. For all $i$ in $[0, |y| - 1]$, create the transitions $r_i \xrightarrow{y_i} r_{i+1}$
12. Add self-loop transition $r_{|y|} \xrightarrow{\Sigma} r_{|y|}$
13. Let $p$ be the longest prefix of $y$ consisting only of the character $y_0$
14. Add self-loop transition $r_{|p|} \xrightarrow{y_0} r_{|p|}$
15. For all $i$ in $[|p| + 1, |y| - 1]$ add transitions $r_i \xrightarrow{\{y_0\} - \{y_i\}} r_1$
16. For all $i$ in $[0, |y| - 1]$ add transitions $r_i \xrightarrow{\Sigma - \{y_0, y_i\}} r_0$
17. For all $c \in \Sigma$, let $m_c$ be the output from Algorithm 1 with $p = y$ and $s = xc$
18. Create the transition $q_{|x|} \xrightarrow{c} r_{|m_c|}$ for all $c \in \Sigma$