THEORETICAL MODELS OF COMPUTATIONS

ARUSH GULIANI

Abstract. This paper aims to establish a basic understanding of theoretical models of computation, beginning with finite automata. Developing formal definitions for different types of finite automata allows for a study of regular languages, or the languages which are accepted by such finite automata. Understanding the properties and computational capabilities of finite automata and regular languages leads into a model of Turing Machines, a much more generalized and powerful model of computation. Still, Turing Machines aren't capable of computing solutions to every problem.

Contents

1. Deterministic Finite Automata

Finite automata were initially proposed as a model to represent brain function, but instead ended up being a crucial building block in the development of theoretical computer science. This model of automata is characterized by its finite set of states, which serves both as its mode of operation and as its primary limitation. A set of inputs takes the automaton from state to state, and at the end of the chain of inputs, it is checked whether the final state of the automaton falls within the set of accepting states. We aim to formally define these finite automata, in order to understand their properties and computational capabilities, including both the type of problems they can solve as well as the ones they cannot.

This explanation of finite automata is still abstract, so we shall present a simple informal example. Consider a bookstore which can have up to 10 books on its shelves at any point in time. The bookstore starts off with a complete set of 10 books, and has a record for every time it sells a book, or restocks on a book. Reading through the record of sales and restocks, and knowing that we started with 10 books, we can determine the amount of books at the bookstore at any given time simply by updating the number of books following each transaction. We can construct a finite automaton which follows this process exactly to determine if the bookstore is empty after a series of transactions.

Date: August 29, 2023.

This automaton would possess 11 states, numbered 0 through 10, corresponding to the number of books present in the bookstore at any given time. The starting state of our automaton would be 10, since this is the number of books we begin with. We would have a transition function which reads the record of sales and restocks and updates our state after each transaction. After every restock, the state would increase by 1, unless we are at 10, in which case it stays constant since we are at maximum capacity. After every sale, the state would decrease by 1, unless we are at 0, in which case it would stay constant since we are at minimum capacity. After reading through the entire record of transactions we are left with a final state, corresponding to the number of books left at the bookstore after all sales and restocks. The bookstore is empty if and only if the final state is 0, and in this case we say that our automaton accepts this transaction log.

We now formally define a Deterministic Finite Automaton (DFA).

Definition 1.1. A DFA is a method of computation which is defined by five parameters, and is typically denoted as

$$A = \{Q, \Sigma, q_i, \delta(q_{in}, a), F\}$$

These parameters are defined as follows:

(1) A non-empty finite set of states denoted by $Q = \{q_0, q_1, q_2, ...\}$
(2) A non-empty finite set of input symbols, denoted by Σ
(3) A starting state $q_i \in Q$
(4) A transition function $\delta : Q \times \Sigma \to Q$
(5) A set $F \subseteq Q$ of all accepting states

The DFA, beginning at the state q_i, reads through the input string one letter at a time. As a DFA reads the input string, it changes between states. In particular, when the DFA is in state q, and reads the character a, it switches to state $\delta(q, a)$. A DFA is said to accept an input string if and only if the final state upon processing the entire string is an element of F. We say a DFA solves a problem if it accepts only those solutions which are answers to the given problem. For example, in the case of the bookstore, the DFA would solve the problem of checking if the bookstore is empty, since it accepts only those transaction logs which would result in an empty bookstore.

Let us construct a DFA which given a non-negative integer, determines the parity of the sum of its digits.

Example 1.2. We will construct a DFA that only accepts strings which sum to be odd.

We can identify our set of input symbols:

$$\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$$

Next, we want to create a set of the possible states. Since the running total can either be even or odd, we need two states. Let $Q = \{q_o, q_e\}$, where q_o refers to an odd running total and q_e refers to an even running total. Since zero is an even number, we know that we start off being even, so:

$$q_i = q_e$$

We next want to pick our set of accepting states. Since we we only want to accept odd totals, we let

$$F = \{q_o\}$$

Finally, we define our transition function, δ. Since we want to update the parity of the running total based on the next digit in the number, reading an even digit should retain the current state, whereas reading an odd digit should switch the current state. Hence, we define δ to have the following values:

- $\delta(q_e, n) = q_e$ if n is an even digit. Else, $\delta(q_e, n) = q_o$
- $\delta(q_o, n) = q_o$ if n is an even digit. Else, $\delta(q_o, n) = q_e$

Now that we have assigned a value to each necessary parameter, we have successfully constructed a DFA which will determine the parity of the sum of the digits of a number.

Let us present some more examples of problems which can be solved by DFAs; the construction of these DFAs is left as an exercise for the reader.

Problem 1.3. *Construct a DFA which can determine whether a string is longer than 5 characters.*

Problem 1.4. *Construct a DFA which can determine whether the length of an input string is a multiple of 3.*

We have seen examples of problems which DFAs can be constructed to solve. This raises the question, what variety of problems can DFAs not solve? We will now prove this to be the case for one such problem.

Theorem 1.5. *There does not exist a DFA which can determine whether a binary string has an equal number of 1s and 0s.*

Proof. For the sake of contradiction, assume there does exist a DFA D which accepts only those strings which contain an equal number of 1s and 0s. Let $n = |Q|$ denote the number of states for this DFA. We know that D must accept a string containing $(n + 1)$ zeroes followed by $(n + 1)$ ones. Let this string be denoted by w.

Since there are $2n + 2$ steps in the computation of w, at least one state must be visited in at least 3 steps.

Let this state be denoted by q_r. Since q_r is visited at least 3 times, there are at least 2 disjoint substrings of w which bring the DFA from q_r back to q_r; let us refer to these substrings as S_1 and S_2. If D is in state q_r and reads S_1 or S_2, then it will once again be in state q_r.

We now seek to prove that either S_1 or S_2 has an unequal number of 1s and 0s. If S_1 has an unequal number of 1s and 0s, then we are done, so we can assume that this is not the case. Since S_1 is non-empty and has an equal number of 1s and 0s, it must fall directly in the center of w, since this is the only way it can contain both 1s and 0s. Since S_1 and S_2 are disjoint, S_2 must either fall to the left or right of S_1, and as a result, will either be composed entirely of 0s or entirely of 1s. We have now proven that either S_1 or S_2 has an unequal number of 1s and 0s.

We now seek to form a new string, which contains an unequal number of 1s and 0s, yet still brings our DFA back to an accepting state. Without loss of generality, assume that S_2 was the substring with an unequal number of 1s and 0s. Let X and Y be the strings such that $w = X + S_2 + Y$, where $+$ is the concatenation operator. Our new string, w', will be defined as the following:

$$w' = X + S_2 + S_2 + Y$$

The number of 1s and 0s in w is equal, and the number of 1s and 0s is unequal in S_2, so the total number of 1s and 0s will be unequal in w'. Furthermore, since

S_2 brings the DFA from the state q_r back to q_r, repeating it an additional time within our input string will have no effect on the final state of the DFA; thus, our DFA will accept w'.

We have now shown our DFA to accept a string which has an unequal number of ones and zeroes, so we reach a contradiction. □

2. Non-deterministic Finite Automota

Let us now try to construct a DFA which accepts only those binary strings ending in "01." This is a surprisingly hard process, since the DFA will never know whether it is approaching the end of the string or not. To make this problem easier to solve, we introduce non-deterministic finite automata (NFAs). These computational devices are identical to DFAs, except for the fact that they can be in several or no states at any given point. Since Q, Σ, Q_i, and F are defined identically to that of DFAs, we will not bother defining these. We instead focus on the transition function, δ, which now possesses the quality of returning a set of output states rather than a singular output state. For an input character a and an input state q_{in}, we have the following:

$$\delta(q_{in}, a) = Q_{out} \subseteq Q$$

The set of states for the NFA upon reading a letter is the union of the transition function outputs for each of the states it is currently in. Let Q_0 be the current set of states of an NFA during computation. Upon reading an input letter a, its new set of states will be

$$Q_1 = \bigcup_{q \in Q_0} \delta(q, a)$$

We say an NFA N accepts an input string w if its final set of states upon reading w, say Q_f, includes at least one member of the set of accepting states. This means that w is accepted if and only if $\exists q \in Q_f$ such that $q \in F$.

Example 2.1. We seek to construct an NFA which only accepts those binary strings which end in 01.

We know the input language for this NFA:

$$\Sigma = \{0, 1\}$$

We next want to define the set of states for this NFA. In particular, we have the state q_0, which is a waiting state, so the NFA is still waiting for the end of the string to begin. Since the NFA will never know for certain that it is approaching the end of the string, it will always remain in this state. However, we also define a state q_1, and this is the state of the NFA which guesses that it has read the second to last digit to be a 0, and it is expecting that a 1 should follow. Since we never know if we are approaching the end of the string, the NFA will always enter q_1 upon reading a 0. Finally, we define the state q_2, a state which corresponds to the NFA guessing that it has read the final 01. As such, the NFA will enter Q_2 whenever it reads a 1 while in state q_1. Crucially, reading any digit while in q_2 yields an empty set of states, since any further digit invalidates the guess that the 01 we have just read is the end of the string. Similarly, reading a 0 while in q_1 returns an empty set of states since this 0 invalidates the guess that we were entering the final 01.

$$Q = \{q_0, q_1, q_2\}$$

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

FIGURE 1. A diagram of the transition function δ, sourced from *Introduction to Automata Theory, Languages, and Computation*, Section 2.3 [1]

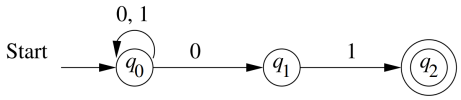

FIGURE 2. An illustration of the behavior of our NFA, sourced from *Introduction to Automata Theory, Languages, and Computation*, Section 2.3 [1]

Naturally, the NFA begins in its waiting state, so:

$$q_i = q_0$$

Furthermore, we want to accept the string if it has read a 01 at the end, so:

$$F = \{q_2\}$$

The transition function, as described above, is given by the Figure 1 where $\emptyset$ corresponds to the empty set, $\rightarrow$ corresponds to the starting state, and $*$ corresponds to an accepting state.

Now that we have defined Q, Σ, Q_i, δ, and F, we have completed the construction of our NFA to determine if a string ends in 01. Figure 2 illustrates the trajectory of the NFA as it reads through the input.

Even though we greatly simplified the solution of this problem by using an NFA, notice that we still could have solved it with a DFA (the details of this construction are left as an exercise). However, this DFA solution would have many more states than the NFA we just constructed, and generally be needlessly complicated. As such, we greatly simplified our computational mechanism by using an NFA. Still, despite the fact that NFAs make computations for certain problems easier, as we have just observed, we still don't know whether NFAs are more powerful than DFAs. To understand whether there is a greater degree of computational capability possible with an NFA than a DFA, we will attempt to write an arbitrary NFA as a DFA.

Theorem 2.2. *Any problem which can be solved by an NFA can also be solved by a DFA.*

Proof. Let us consider an arbitrary NFA A given by:

$$A = \{Q, \Sigma, q_i, \delta(q_{in}, a), F\}$$

We want to construct a DFA, B, which accepts exactly the same set of strings as A. Σ will remain the same for B, since it reads the same alphabet. We now define P to be the power set of Q. Every possible set of states which A could be in will be included in P. We now define X to be the subset of P which includes every set with at least one element of F. This will be our set of accepting states for B. For the starting state for B, we have $\{q_i\}$, the set containing the starting state of A. Finally, we will construct a new transition function, denoted δ', and defined as follows:

$$\delta'(S \subset Q, a) = \bigcup_{q \in S} \delta(q, a)$$

This transition function will move between sets of states exactly in accordance with A. Now that we have constructed each parameter of B, we can define it to be the DFA given by:

$$B = \{P, \Sigma, \{q_i\}, \delta', X\}$$

By its construction, B will behave identically to A, and thus accept the same set of strings.

□

3. Regular Languages and Expressions

So far, we have discussed DFAs and NFAs, two models of computation with differing modes of operation but identical computational capabilities. We have considered the type of problems which these automata can solve, and ones they cannot solve. To further generalize which type of problems are capable of being solved by these finite automata, we want to develop a system of regular languages, which are defined to be the sets of strings which are accepted by some finite automata. Looking at regular languages, we begin with the question: what pattern unites these regular languages? What properties must they follow and what does this tell us about finite automata?

Definition 3.1. A language is a set of strings built from letters within a given alphabet, Σ. The language of a finite automata is defined to the set of strings which it accepts. For an NFA or DFA A, $L(A)$ notates the corresponding language. Any language which has a corresponding DFA is considered to be a regular language.

We want to define a few operators for languages.

(1) The union of two languages, $L_1 \cup L_2$.
(2) The concatenation of two languages, denoted as $L_1 \cdot L_2$, is the set of all strings $w = xy$ for some $x \in L_1$ and some $y \in L_2$.
(3) The closure of a language, denoted as L_1*, is the set of all strings, w, such that for some non-negative integer n, $w = x_1x_2...x_n$, with $x_i \in L \; \forall i$.

For example, if:

$$L_1 = \{0, 1\}$$
$$L_2 = \{a, b, c\}$$

Then:

$$L_1 \cup L_2 = \{0, 1, a, b, c\}$$
$$L_1 \cdot L_2 = \{0a, 0b, 0c, 1a, 1b, 1c\}$$

$$L_1* = \text{the set of all binary strings}$$

We now seek to separately define regular expressions, which will help us talk about regular languages in a more algebraic manner.

Definition 3.2. We will define regular expressions inductively. We begin with our base case, by establishing the three cases which we consider to be regular expressions by definition. For any symbol x, $\mathbf{x}$ is a regular expression denoting the singleton language containing that symbol. We write this as

$$L(\mathbf{x}) = \{x\}$$

A special case of the singleton regular expression is ϵ, a regular expression denoting the language $\{\epsilon\}$, or the language containing the empty string. The final base case of our regular expression is $\emptyset$, which denotes the empty language.

From here, we inductively define that for regular expressions X and Y, the following is also true:

(1) $X + Y$ is a regular expression, and the language of $X + Y$ is $L(X) \cup L(Y)$
(2) XY is a regular expression, and the language of XY is $L(X) \cdot L(Y)$
(3) $X*$ is a regular expression, and the language of $X*$ is $L(X)*$

Now that we have formally defined regular expressions and their arithmetic, we can now use them to talk about languages. Crucially, we have a relation between regular expressions and regular languages.

Theorem 3.3. *A language is regular if and only if it is represented by some regular expression.*

The proof of this theorem is beyond the scope of this paper, but it generally functions by inducting on the arbitrary regular expression to build a corresponding DFA, and in reverse by inducting on the the paths between states to build a regular expression.

Using the models we have built up, we now want to prove some properties of regular languages, since this will better give us an idea of the kinda of problems which may be solved by finite automata.

Theorem 3.4. *If L is regular, then L^R is regular, where L^R is the set of all strings in L reversed.*

Proof. Let L be some arbitrary regular language. By Theorem 3.3, there exists some regular expression, say E, such that $L(E) = L$. To show that L^R is a regular language, it suffices to show that there exists some regular expression E^R such that $L(E^R) = L^R$.

We will build up E^R by considering the form of the expression E and determining its reversal, repeating this process through induction on the size of E. We now want to show the value of E^R depending on the different possible forms of E.

- In the base case that E refers to a singleton set or an empty set, the corresponding language is a reversal of itself, and so we set $E^R = E$, since this would mean that $L(E^R) = L(E) = L = L^R$.
- If E is of the form $E = A + B$, then $E^R = A^R + B^R$. This is because the $+$ operator refers to a union, and the reversal of the union of two languages is simply the union of the reversal of both languages. For example, if

$L(A) = \{100, 101\}$ and $L(B) = \{001, 010\}$, then, $L(A^R) = \{001, 101\}$ and $L(B^R) = \{100, 010\}$. This would give us the correct value for E^R:

$$E^R = A^R + B^R = \{001, 101, 100, 010\}$$

- If E is of the form $E = AB$, then $E^R = B^R A^R$. This is because when dealing with the concatenation of two languages, the reversal of the concatenation will be the concatenation of the reversal of each language, but in opposite order. Using the same A and B as in the previous example, we calculate the correct value for E^R in this case as well:

$$E^R = B^R A^R = \{100001, 100101, 010001, 010101\}$$

- The case that E is of the form $E = A*$ follow a similar logic to the previous two forms, and the specific math is left as an exercise to the reader.

  Now that we have inductively reverse engineered a regular expression E^R, such that $L(E^R) = L^R$, we are done.

□

Theorem 3.5. *The complement of a regular language with respect to an alphabet is a regular language.*

Proof. We pick some arbitrary regular language L. Since L is regular, there exists a DFA whose set of accepted strings is L. Let this DFA A be given by:

$$A = \{Q, \Sigma, q_i, \delta(q_{in}, a), F\}$$

Let X be the language which is the set of all strings comprised of alphabet Σ. We want to show that $L^c = X \backslash L$ is a regular language. We construct a DFA A' given by:

$$A' = \{Q, \Sigma, q_i, \delta(q_{in}, a), Q \backslash F\}$$

Since the accepting states of A' is all of the rejecting states of A, A' will accept every string which was not accepted by A, meaning that its set of accepted strings is exactly L^c. Since L^c is given by a DFA, it is a regular language.

□

The final property of regular languages we seek to establish is known as the Pumping Lemma, and is essentially a more general version of the phenomenon we observed in Theorem 1.3, which stated that no regular language consists of only those binary strings which have an equal number of 1s and 0s. By establishing this Pumping Lemma, we will have a more direct way of observing of which sets of strings can be regular languages, and by extension, which problems can be solved for by finite automata.

Theorem 3.6. *Pumping Lemma: Let L be a regular language. There exists an integer n such that for any string $x \in L$ with $|x| \geq n$, x can be broken down into the concatenation of 3 strings, written as $x = abc$, such that:*

$$b \neq \epsilon$$

$$|ab| \leq n$$

$$\forall k \geq 0, ab^k c \in L$$

Where b^k represents the repeated concatenation of the string b, and $|x|$ represents the length of string x.

Theorem 3.6 follows a proof structure parallel to that of Theorem 1.5. As such, this proof is left as an exercise to the reader.

Let us walk through an example of the applications of this Pumping Lemma.

Example 3.7. Prove that there is not a regular language which consists only of prime-length strings.

Proof. Assume for the sake of contradiction that there exists a regular language L which consists only of strings of a prime number length. We pick the least prime number, denoted p, which is greater than the n given by our Pumping Lemma. We pick a string from our language with p digits, and let this string be denoted x. We break x down into the concatenation of 3 strings, as given by the Pumping Lemma

$$x = abc$$

We know that $|b| \neq 0$ by the pumping lemma, and we know that $|abc| = p$ by definition. We now form a new string:

$$x' = ab^{p-|b|}c$$

By the Pumping Lemma, we know that $x' \in L$. We can also calculate that:

$$|x'| = (p - |b|) + |b| * (p - |b|) = (|b| + 1)(p - |b|)$$

Since we have just factored $|x'|$, we know that x' does not have a prime length, but that it is a member of L. Thus, we reach a contradiction. □

4. Turing Machines

We have looked at finite automata and their limitations, including problems which they cannot solve, such as whether a string has a prime number of digits, or if a string has the same number of 1s and 0s. We will draw a contrast to this limited computational model by briefly considering Turing Machines, a more abstract yet powerful model of computation. This model of computation includes the ability to solve every problem that is solvable by a modern computer. We will consider the computational capabilities and logical limitations of these Turing Machines.

Definition 4.1. Turing Machines are a model of computation which possess the ability to execute any algorithm. Their functionality can be conceptually understood through the idea of a machine reading and writing over an infinite piece of tape containing inputs. Specifically, consider a piece of tape broken into sections, each of which can contain a single symbol, or be blank. This piece of tape has a distinct starting point, but continues infinitely in the other direction. There is a finite amount of input symbols on this infinite piece of tape, beyond which are exclusively blank symbols. Also, like finite automata, the Turing Machine is in one of a finite set of states at any given point in time. The Turing Machine is looking at exactly one section on the tape at any given point in time, starting with the first section, and has the ability to:

(1) Read the section it is currently on.
(2) Erase or re-write the symbol on the section it is currently on.
(3) Move to the next section forward or the previous section.
(4) Change its current state.

There are unique states, known as the accepting and rejecting states, which upon entering, cause the Turing Machine to accept or reject the input string, respectively. A Turing Machine can accept, reject, or enter an infinite loop when it reads an input string.

Traditionally, and formally, Turing Machines are denoted by a tuple notation similar to that which we used for finite automata. This tuple notation definition varies by source, but includes things such as the input alphabet, a set of states, a starting state, an accepting state, a rejecting state, a dedicated blank symbol, and a transition function which dictates the rewriting of symbols and the movement between states. However, for the purposes of this paper, we will not use the formal tuple definition, and instead refer to these Turing Machines simply using the notation $A(a_1, a_2, ...)$, where A is the machine itself, $a_1, a_2, ...$ are the inputs which are written on the tape, and $A(a_1, a_2, ...)$ denotes the output of the Turing Machine when reading the given inputs, being accepting, rejecting, or looping. Although we will not prove why this model of Turing Machines can be used to form any possible algorithm, we know this to be the case by the Church-Turing Thesis.

With this model of Turing Machines established, we pose the question: is it possible to construct a Turing Machine for any language, such that only strings within that language will be accepted? In other words: can Turing Machines solve every problem? Despite this model of Turing Machines encapsulating the abilities of every computer ever made, we know that there exist some problems which cannot be solved by even this device.

Theorem 4.2. *Not every problem is solvable by a Turing Machine.*

Proof. For any Turing Machine T, let $w(T)$ represent the blueprint of the Turing Machine T written in the corresponding language. For this proof, we will also treat any situation in which a Turing Machine infinitely loops as it rejecting the given input.

Suppose every problem was solvable by a Turing Machine. We could construct a Turing Machine A which would take in two sets of inputs: an arbitrary set of inputs k, followed by a blueprint for a Turing Machine, $w(F)$. A would accept only those pairings of Turing Machines and symbols such that F accepts the input k.

We will now contradict our assumption by proving that A cannot exist.

We begin by constructing a Turing Machine B. B takes in the blueprint for a Turing Machine, $w(F)$, and works in the following way:

$B(w(F))$ rejects if $A(w(F), w(F))$ accepts, and $B(w(F))$ accepts if $A(w(F), w(F))$ rejects. In other words, B accepts only those Turing Machines which reject their own blueprints.

We now disprove the existence of A by showing that $A(w(B), w(B))$ fails to work in the ways we would expect, and instead reaches a logical contradiction.

If $A(w(B), w(B))$ is accepting, then B must accept input $w(B)$. However, this can only happen if $A(w(B), w(B))$ rejects. Similarly, if $A(w(B), w(B))$ rejects, then B must reject input $w(B)$. However, this can only happen if $A(w(B), w(B))$ accepts. In both cases, we reach a contradiction, and this disproves the existence of A.

□

Compared to the limited reach of finite automata, Turing Machines have incredible scope, and the fact that we are able to find an unsolvable problem for them

means that we have proved a limit for computation. Beginning with finite automata and their simple ability to check whether a string ends in 01, we now look at the very limits of possible computational capabilities, and know that there is in fact a a limit to what computers can, and will be able to do.

Acknowledgments

It is a pleasure to thank my mentor, Marc de Fontnouvelle, for exposing me to the world theoretical computation, and for guiding me through my reading of *Introduction to Automata Theory, Languages, and Computation*[1]. He introduced me to many different concepts and theorems in the world of Automata and Complexity theory which led me to pursue this paper topic, and this paper would not have been complete without his continuous support.

References

[1] Hopcroft, John E., et al. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[2] Kozen, Dexter. *Automata and Computability.* Springer, 1997.