An Introduction to the Theory of Computation

Ryan Chao and Yakir Propp

April 2023

1 Introduction

For this paper, we will be discussing three classes of automata: Finite Automata, CFG & PDAs, and Turing Machines. We will be providing the definition of all these automata, as well as their properties and uses. We will go into depth on the equivalence theorems between different subtypes of the same class.

2 Finite Automata

Let's discuss the building blocks of Computation! Finite Automata are the simplest type of automata, creating the foundation for our ideas of Context Free Grammars (CFGs), Pushdown Automata (PDAs), and Turing Machines (TMs). An extremely high level definition of Finite Automata is that they are a theoretical machine that takes in an input string, and then either accepts or rejects it.

There are two types of finite automata: Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA). We will find that these two are actually equivalent computation models.

2.1 Deterministic Finite Automata (DFA)

As explained earlier, a Finite Automata takes in an input string and either accepts or rejects it. Now how does a Finite Automata know whether to accept or reject a string? Let us answer this question in the case of DFAs.

Informal Definition 2.1. A DFA has a collection of states, which it can enter based on the input string. There are transition functions that say which state the machine should enter depending on what the contents of the string is. By the time the DFA finishes reading the string it will have gone into either an accept or not an accept state, meaning it would have either accepted or rejected the string.

Let's take a look at an example of a DFA.

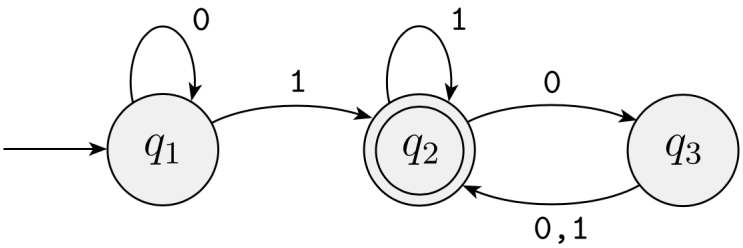


Figure 1: Finite Automata G_1

Example 2.2. Say we were to input a string, 1100 for example. When we first take in 1100 it goes straight to the start state denoted by an arrow coming in from nowhere, q_1. Looking at the arrows, which represent the transition functions, we see that starting from q_1 we have two possible actions: if the first character is 0, stay at q_1, and if the first character is 1, move on to q_2, which happens to be the accept state notated by the double circle. We read the first character, a 1, so we move on to q_2. Then at q_2 we read the next character, another 1, so according to the transition functions we stay at q_2. Then as we move on through the string we read a 0, making us go to q_3 and finally a 0 again, causing us to end up in q_2, an accept state, which means our machine G_1 accepts the input string 1100. Had the machine not ended up in the accept state then we would have rejected it.

Formally, we can describe a DFA as such:

Definition 2.3 ([1, Definition 1.5])**.** A ***Finite Automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \to Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

In our example, $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, the accept state q_0 is q_1, and $F = \{q_2\}$. We also have that $\delta(q_1, 0) = q_1$, $\delta(q_1, 1) = q_2$, and so on.

Definition 2.4. A *language* is the set of strings that a finite automata accepts.

Definition 2.5. If the the language of machine M is A, i.e. $L(M) = A$, then we say M *recognizes* A. For any set B where $B \neq A$, M does **not** *recognize* B.

Definition 2.6 ([1, Definition 1.16])**.** A *regular language* is a language which is recognized by some Finite Automata. Regular languages are vitally important to our study of Automata, as they can show us the limits of state machines in accepting input.

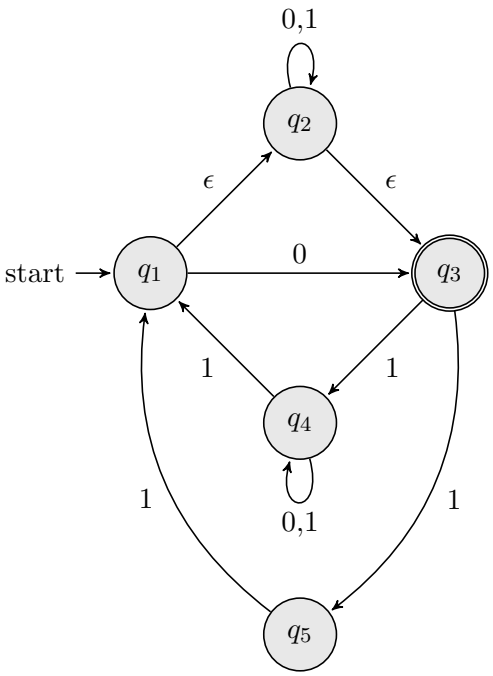
The type of Finite Automata that is described in this section is called a Deterministic Finite Automata (DFA), and this is because there is only one path you can follow with each string. If there is such a thing as a Deterministic Finite Automaton, does this mean that there could be a Nondeterministic Finite Automaton (NFA)? I'm glad you asked! Yes, there is! We will discuss this in the next subsection.

2.2 Nondeterministic Finite Automata (NFA)

The key difference between a DFA and NFA is that an NFA can have multiple paths for an input.

Informal Definition 2.7. A Nondeterministic Finite Automata (NFA) is a Finite Automata in which there may be multiple or zero transitions from any state for any given letter of the *alphabet*.

Here is an example of an NFA:



We can see from this example that NFAs exhibits many properties that DFAs don't. First of all, there may be multiple or zero of the same type of arrow going out from each state. From example, state q_3 has two 1-arrows going out from it, but zero 0-arrows. In addition, there is a new type of arrow introduced, called an epsilon arrow. An epsilon arrow is an arrow which transitions while taking in the epsilon symbol i.e no input. Thus an arrow from q_i to q_j can be thought of as putting us in states q_i and q_j simultaneously. We compute by moving along all possible arrows, rather than just the one path for a DFA. If any of the states that we could end in after processing the input is an accept state, we accept.

The following is the formal definition of an NFA.

Definition 2.8 ([1, Definition 1.37])**.** A *Nondeterministic Finite Automaton* (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states
2. Σ is a finite alphabet
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function
    - Σ_ϵ notation just means $\Sigma \cup \epsilon = \{0, 1, \epsilon\}$. In other words, ϵ arrows are available as well as 0, and 1 arrows.

- The ϵ arrow is a transition function which takes in no input symbol and moves to the corresponding state.
- $\mathcal{P}(Q)$ is the Powerset of $\{Q\}$, which is the set of all subsets of Q (Q is considered a subset of Q as is $\emptyset$).

4. $q_0 \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states.

Note that every DFA is also an NFA, since having exactly one transition for every letter in every state counts under the definition of a NFA. While DFAs are the simplest kind of automata, it is often more natural to model a situation using NFAs. This definition of an NFA should be very easy to understand based on the definition of Deterministic Finite Automaton. The only condition that is changed is condition 3. The range is now the Powerset of Q, since the output of any state will be a set of states (this includes the null), rather than just a single state. This definition motivates the following theorem:

Theorem 2.9 ([1, Theorem 1.39])**.** Any NFA has an equivalent DFA, which means that they recognize the same language.

Proof. For a NFA with states Q, and alphabet Σ, we can construct a DFA whose states is $R := \mathcal{P}(Q)$. Then, for each letter of the Σ for every set in R, R_i, take the union of the delta functions for Σ for the subsets of R_i. Now write a transition between R_i and this new set. This is a DFA, since there is example one transition for every R_i. The union represents the possible states that the automaton could end up on after one transition, starting from a certain number of states (represented as a set of those states). □

This theorem is a very powerful result in the theory of computation. Since DFA's have been shown to be equivalent to NFA's, then the following result about regular languages (a property of DFA's) can be proven using NFA's: regular languages are closed under the union, concatenate and star operations. In other words, for these operations, if there exists an NFA which recognizes the input(s) of the operation, then there also exists an NFA which recognizes the output of the the operation. First we will show the construction of the NFA which recognizes the union of two regular languages.

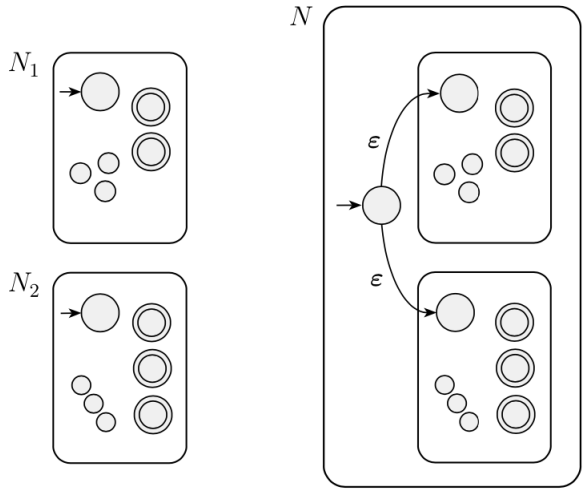


Figure 2: Construction of an NFA N to recognize $A_1 \cup A_2$
[[1, Theorem 1.45]]

Example 2.10. We have regular languages A_1 and A_2 and our goal in to prove that $A_1 \cup A_2$ is regular. We can take two NFAs, N_1 and N_2 for A_1 and A_2, and then combine then into one new NFA N. We combine N_1 and N_2 by creating two ϵ arrows coming from a new start state and pointing to each respective start state of N_1 and N_2 as shown in Figure 2. This nondeterministically simulates a new NFA that recognizes $A_1 \cup A_2$, meaning that $A_1 \cup A_2$ must be regular.

Next we will show the construction of an NFA, which represents the concatenation of two languages. Concatenation example: $\{1, 4, 5\} \circ \{2, 3, 6\} = \{12, 13, 16, 42, 43, 46, 52, 53, 56\}$

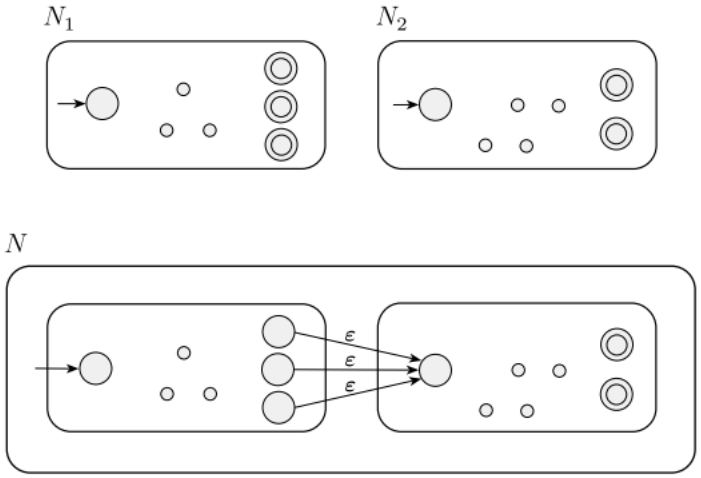


Figure 3: Construction of an NFA N to recognize $A_1 \circ A_2$
[[1, Theorem 1.48]]

As we can see, this new machine, N, works by first simulating N_1. Only the strings that are accepted by N_1 get passed into N_2 via the epsilon arrows, which must then connect to a string

which is accepted by N_2. In this way, we have constructed an NFA which recognizes all pairwise concatenations of accepted strings of N_1 and N_2, which is exactly $A_1 \circ A_2$.

3 Context Free Grammar (CFG) and Pushdown Automata (PDA)

In this section, we will introduce Context Free Grammars (CFGs) and Pushdown Automatas (PDAs), and show that they are equivalent.

3.1 Context Free Grammars (CFG)

A Context Free Grammar (CFG) is as a more powerful method of describing languages, which were introduced in the previous section. CFG's are associated with Context Free Languages (CFL). We can define our grammar as a set of substitution rules. Each rule comprises of a variable pointing to a string, and each string can be made up of variables and other symbols which are called terminals. We also have one variable which is called the start variable and usually resides at the top of our grammar definition. So, why are these CFGs equivalent to languages?

Well, we can use our substitution rules to create all the strings that would otherwise be described by a language. We can perform a sequence of substitutions to derive a string that is recognized by an Automata. For example, lets say we had a CFG G_1 which was defined as:

$$\begin{aligned} A &\to 0A1 \\ A &\to B \\ B &\to \# \end{aligned} \tag{1}$$

To derive the string 000#111 in Grammar G_1 we could do

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111 \tag{2}$$

We can say that all strings which are made in this manner are called the language of grammar G_1, or L(G_1)
Now for a formal definition of a CFG, we can say that:

Definition 3.1 ([1, Definition 2.2])**.** A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the variables
2. Σ is a finite set, disjoint from V, called the terminals
3. R is a finite set of rules, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable

3.2 PDA

A ***Pushdown Automata*** is a type of Finite Automaton that uses a data structure called a stack. A stack uses the LIFO philosophy (last in first out). Imagine a stack of plates at a restaurant sitting in bin. The waiter can add plates to the top and "push down" which would move each of the plates one plate-height downwards, or remove a plate from the top of the stack. The advantage of the ***Pushdown Automata*** over regular DFAs and NFAs is it has a "memory," or a place to

store in Each plate can take on values contained with the **stack alphabet**, Γ, which can consist of any symbols. The transition function has to take into account as well as the states machine, symbols being deleted from or added to the stack. This leads us to the following definition.

Definition 3.2 ([1, Definition 2.13])**.** A ***Pushdown Automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F, are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function
5. $q_0 \in Q$ is the start state
6. $F \subseteq Q$ is the set of accept states.

As we can see, this definition is similar to that of NFAs (here we are considering NPDAs), however, the NPDA (nondeterministic pushdown automaton) must take into account how the stack alphabet changes in step 4. If for example, we were to add the symbol, b, during the transition, the stack would go from ϵ to b. If we wanted to remove the top symbol b, the stack goes from b to ϵ. We can also swap out the top of the stack in the transition by going from symbol a to symbol b. This can be thought of as a two-step process of removing a and adding b.

Theorem 3.3 ([1, Theorem 2.20])**.** A language is context free iff some ***Pushdown Automaton*** recognizes it.

The proof of this theorem can be broken down into two parts

Lemma 3.4 ([1, Lemma 2.21])**.** If a language is context free, then some ***Pushdown Automaton*** recognizes it.

Proof. Let A be a CFL with language L. By definition, A must have a CFG, G, generating it. The goal is to construct a PDA, P, which simulates the rules of G. If we are to accept when the terminal symbols have been reached, then we will accept an element of L. P works as follows:

1. Place the marker symbol \$ and the start variable on the stack
2. Repeat the following steps forever.
   (a) If the top of stack is a symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule. If not, proceed to step 2.
   (b) If the top of stack is a terminal symbol, t, read the next symbol from the input and compare it to t. If they match, go back to step a. If they do not match, reject on this branch of the nondeterminism.
   (c) If the top of stack is the symbol \$, enter the accept state. Doing so accepts the input if it has all been read.

□

The proof is fairly easy to understand. We convert our start variable into terminal variables, bit by bit. We verify if the terminals match the strings in the language that we want to recognize, and keep converting until they are all terminals.

Lemma 3.5 ([1, Lemma 2.22])**.** If a ***Pushdown Automaton*** recognizes some language, then it is context free.

We first have to simply P to construct the grammar which generates the language. Here are the following three simplifications:

1. It has a single accept state, q_{accept}.
   - This can easily done by attaching ϵ arrows from all of the accept states from the PDA, to a new state, which will become the accept state. This PDA would obvious recognize the exact same language and thus be equivalent.
2. It empties its stack before accepting.
   This could easily be implemented, and would not change what languages the PDA accepts.
3. Each transition either pushed a symbol onto the stack (a *push* move) or pops on of the stack (a *pop* move), but it does not do both at the same time.
   The way this is done is we replace the transitions that pop and push simultaneously with a two-step transition between states. For transitions which neither pop or push, we create a two-step transition which pushes and then pops and arbitrary symbol.

Proof. Say that $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{accept}}\})$ and construct G. The variables of G are $\{A_{\text{pq}} | p, q \in Q\}$. The start variable is $A_{q_0, q_{\text{accept}}}$. Now we describe G's rules in three parts.

1. For each p, q, r, $s \in Q$, $u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ), put the rule $A_{\text{pq}} \to aA_{\text{rs}}b$ in G.
2. For each p, q, $r \in Q$, put the rule $A_{\text{pq}} \to A_{\text{pr}}A_{\text{rq}}$
3. Finally, for each $p \in Q$, put the rule $A_{\text{pq}} \to \epsilon$ in G.

□

The way this works is that since the machine either pops or pushes a symbol for each transition, between two states p and q, the symbol first pushed could be popped getting to state q, or in-between, represented by rules 1 and two 2 respectively.

4 Turing Machine (TM)

If you like computers, this is the section for you! Turing Machines are the first version of modern day computers, and are what this section will be discussing.

4.1 Deterministic Turing Machine (DTM)

The ***Turing Machine***'s significance and importance is best highlighted by something called the ***Church-Turing Thesis***, which states that

Definition 4.1. Any real-world computation can be translated into an equivalent problem involving a Turing Machine

At a high level this is saying that anything a human could do a ***Turing Machine*** could as well, hence our computers which are so darn good at, well, computing. ***Turing Machines*** take us all the way back to Finite Automata, except with a few main differences. First and foremost, they are infinite, meaning that they can store infinite memory (if only my devices did). The way they accomplish this is by having an ***infinite tape***, and on this tape there are input strings which the Turing Machine can read. Turing Machines keep track of where they on this tape by having a ***tape head***, which it can move around the strings. To put this all together, we say that:

Definition 4.2 ([1, Definition 3.3])**.** A ***Deterministic Turing Machine*** is a 7-tuple (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}), where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the *blank symbol* $\sqcup$,
3. Γ is the *tape alphabet*, where $\sqcup \subseteq \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \subseteq Q$ is the start state,
6. $q_{accept} \subseteq Q$ is the accept state, and
7. $q_{reject} \subseteq Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

The transition function, δ, is saying that the tape head can move either left (L), right (R), or not move at all (N). We can see this better illustrated if we follow an example computation involving a Turing Machine. Our example Turing Machine going to look exactly how we described, and will have an input string on the tape ready to go

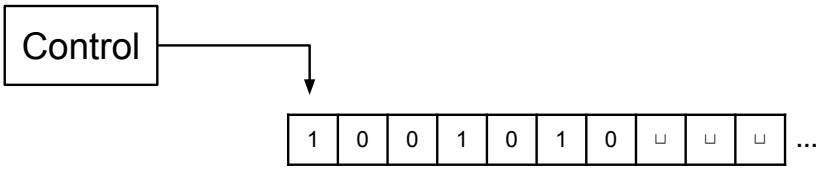


Figure 4: Example Turing Machine

If we were to really fully draw out a Turing Machine it would be a mess of states and transition arrows, but to better understand and visualize what a Turing Machine is we have made a "Control" box. The "Control" box is meant to symbolize the states and transition functions that would control the tape head and mechanics of the machine. The arrow is representing the tape head, and we can see that the head is pointing to the first symbol in the input string "1001010", a "1". Also on the tape are the blank symbols ⊔, and after the tape there is a "..." reminding us that the tape is infinite. Now that we've set up the schematic of a Turing Machine let's actually solve a problem with it.

Example 4.3. Suppose we had to find whether or not an input had an equal number of 1's and 0's. Well, to do that we'd have to create a Turing Machine to compute it, and to build a Turing Machine to compute it we'd have to change around what is in the "Control" box. We will do this by writing an algorithm that that describes a Turing Machine M_{eq}. M_{eq} should decide the language $A = \{w \in \{0,1\}^* \mid \text{where w contains an equal number of 1's and 0's}\}$.

We can now say that M_{eq} = "On input string w:

1. Scan left to right across the tape
2. Replace the first 1 with an "x" and the first 0 with an "x"
3. If the entire string is full of x's, accept
4. If all non-x symbols are the same, reject
5. Return the head to the left end of the tape
6. Go to stage 1"

Our first step, scanning left to right across the tape, is simply reading the input string so that we may manipulate it how to we want so as to achieve our goal. The idea behind the next few steps is that if we keep replacing a 1 with an x and a 0 with an x each iteration we will hopefully end up with a string entirely filled with x's, meaning we had an equal number of 1's and 0's. If there were a few more 1's than 0's (or 0's than 1's) then we'd be left with a string filled with x's and a few 1's (or 0's) at the end. In that case we'd have to reject the string because it would not have an the same number of 1's and 0's. Step 4 is done is by creating a state for the first non-x symbol, and from there, keep track of whether the subsequent non-x symbols differ while traversing the tape. Lastly, iterate by performing the last two steps, returning the head to the beginning and then repeating until we accept or reject.
Running M_{eq} on the input string that we saw earlier, "1001010", we see that we end up with an extra 0, meaning there was not an equal number of 1's and 0's so we would reject it.

The Turing Machine described in this example and the previous definitions all fall under the category of "Deterministic" Turing Machines, however as we will see in the next section there are actually many other variants of Turing Machines.

5 Variants

In this section we will discuss two varients of Turing Machines, the Multitape Turing Machine and Nondeterministic Turing Machine. We will go into more depth on their relationship and equivalence as well.

5.1 Multitape Turing Machine (MTM)

The Multitape Turing Machine and Deterministic Turing Machine are actually very similar, except, as the name suggests, the MTM has multiple tapes whereas the Deterministic Turing Machine has only one. Each of these tapes has it's own tape head, which the machine can operate simultaneously. This means we have to change up our old transition function to accommodate for the tapes. Our new one would be

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k.$$

where k is the number of tapes. We raise Γ and $\{L, R, N\}$ to the power of k to show movement over each of the k tapes. If we were to picture a Multitape Turing Machine it would look as you would imagine, exactly like a Turing Machine just with multiple heads pointing to multiple tapes.

Figure 5: Example Multitape Turing Machine

The Multitape Turing Machine M above has three tapes, each holding a different input string and each with their tape head in a different position.
Looking at Multitape Turing Machines it may be easy for you to make the assumption that they are more powerful than Deterministic Turing Machines, however that is not the case! They are actually equivalent!

Theorem 5.1 ([1, Theorem 3.13])**.** Every Multitape Turing Machine has an equivalent single-tape (Deterministic) Turing Machine

Proof. The idea begind this proof is that if we can simulate a Multitape Turing Machine on a single-tape Turing Machine then we can therefore say that they recognize the same language and for that reason they would be equivalent. The way we do this is best represented by a picture, as shown below:

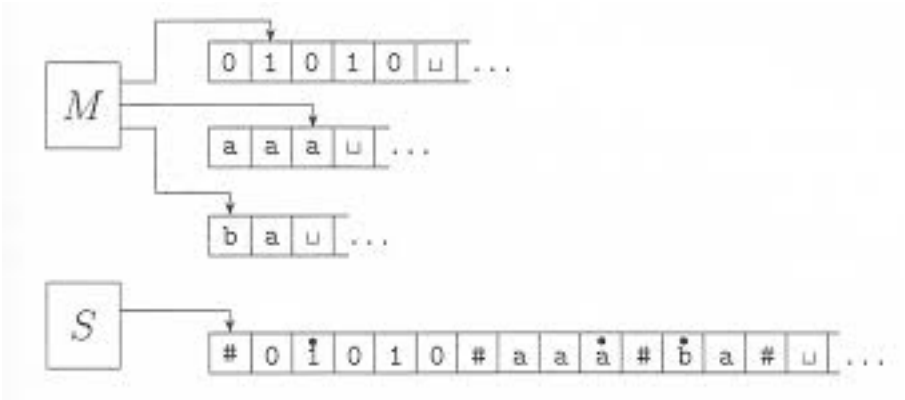


Figure 6: Multitape Turing Machine simulated on a single-tape Turing Machine

We can see the same Multitape Turing Machine M from Figure 4 along with a single-tape Turing Machine S. S is holding each of the tapes from M and is seperating them by using the "#" symbol. We can also notice that there is a dot above some of the symbols. This is how S is keeping track of where each of the tape heads from M were. It is in this way that we can simulate a Multitape Turing Machine M on a single-tape Turing Machine S, hence showing how Deterministic and Multitape Turing Machines are equivalent. □

5.2 Nondeterministic Turing Machine

Nondeterministic Turing Machines are also quite similar to Deterministic Turing Machines except that they have nondeterminism, which was introduced in Finite Automata. To incorporate that idea of nondeterminism in Turing Machines we must change the transition function of the Deterministic Turing Machine such that the Nondeterministic Turing Machine can move to any of the states available and the tape head can have many possibilities for where it can go next. We do this by making it the Powerset of $Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$

$$\delta : Q \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^k \times \{L, R\}^k)$$

This should remind you of the difference between NFA's and DFA's. The next thing we can say about Nondeterministic Turing Machines and Deterministic Turing Machines should as well! It may surprise you, but Nondeterministic Turing Machines, Like Multitape Turing Machines, just as powerful as Deterministic Turing Machines.

Theorem 5.2 ([1, Theorem 3.16])**.** Every Nondeterministic Turing Machine has an equivalent Deterministic Turing Machine

Proof. We prove this by simulating a Nondeterministic Turing Machine on a Multitape Turing Machine. Once we do this we can say that Nondeterministic Turing Machines are equivalent to Deterministic Turing Machines because of Theorem 5.1 where we stated that Deterministic Turing Machines are equivalent to Multitape Turing Machines. The way we do this simulation is by representing a Nondeterministic Turing Machine N as a Multitape Turing Machine D having 3 tapes. One tape, called the input tape, is where we always store the input string. A second tape, called the simulation tape, is where we store a copy of N's tape when it's on some branch of its nondeterministic computation. And on the third and final tape, called the address tape, keeps track of where D is in N's nondeterministic computation. The simulation is illustrated below

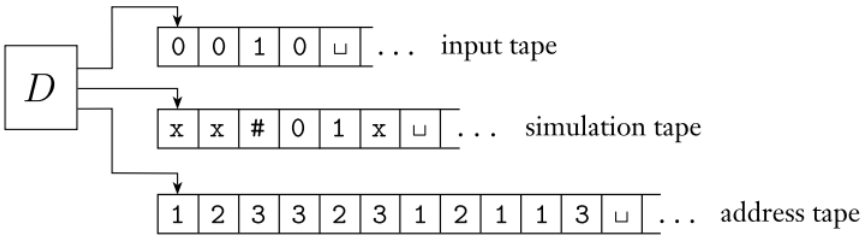


Figure 7: Nondeterministic Turing Machine N being simulated on a Multitape Turing Machine D

Therefore we can say that Nondeterministic Turing Machines and Multitape Turing Machines are equivalent, hence proving that Nonderministic Turing Machines and Deterministic Turing Machines are equivalent. This mirrors the equivalence relationship relationship between nondeterminism and determinism seen earlier in Finite Automta. □

References

[1] Sipser, M. (2013). *Introduction to the theory of Computation* (3rd ed.). Course Technology Cengage Learning.