EXPLORING NONDETERMINISM THROUGH DFS AND BFS

SHERRI WU (MENTOR: KATHERINE TAYLOR)

Abstract. This paper explores the foundational concepts of the Theory of Computation, with a particular focus on nondeterminism and how it can be modeled and understood using depth-first search (DFS) and breadth-first search (BFS). The first part of the paper introduces Finite Automatons, and explains how depth-first-search plays an important role in the processing of non-deterministic finite automaton. The second part of the paper briefly introduces Turing Machine, and introduced another way of exploring graph and process input with breadth-first-search.

1. Introduction

Theory of Computation is a branch of Theoretical Computer Science and Mathematics. It helps us to understand the three questions mentioned in the Abstract: What kind of questions are solvable through computation? How many resources are needed? What models are needed? These three questions are deeply related. To answer the first question, we have computability, which is a branch that helps us figure out whether a question is computable. After that, we have Complexity Theory to help us estimate how much time and memory space we need to solve this question. The Automata Theory and Formal Language gave us highly structured tools and models to support the other work.

Although modern programming tools are widely used, the theory of computation remains essential - not only because it predates these tools, but because it provides the theoretical foundation of designing more efficient algorithms. For example, Automaton are widely used in machine tools and are fundamental in the design of compiler. Moreover, for some certain tasks, the regular expressions derived from Automata theory can outperform the regular expression engines in popular scripting languages like Perl, Java, Python, etc. The graph below illustrates the difference in performance when matching a string using Perl versus with using an NFA.

In the later section of this paper, we will introduce the complexity theory with some examples of regularly used algorithms while solving the Shortest Path Problem. This part will include specific algorithms and analysis on their complexity.

2. Automaton and Languages

2.1. **Finite Automaton.**

The first thing we need is a computational model. To start we need to understand the simplest model, called the finite state machines or finite automaton. Let us first think about a vending machine.

Imagine there is a vending machine that sells snack A for 1 dollar and snack B for 2 dollars, and only accept coins of 1 dollar or 50 cents. So, if you press button A, the machine would go to the state A0 meaning it is waiting to receive 1 dollar, and if you put in 1 or two.5 coins, the machine

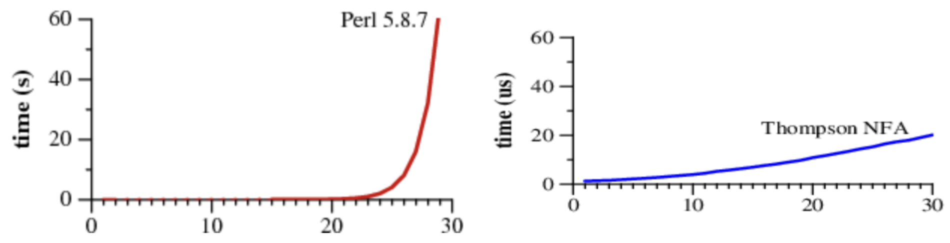
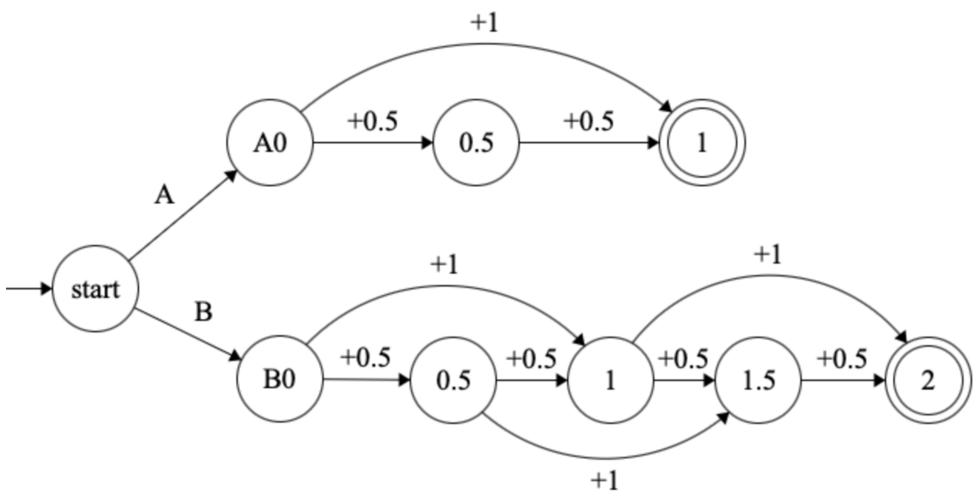


FIGURE 1. As shown by Cox [2], the Thompson NFA is drastically faster.

will go to the state 1 which indicates that you already put in 1 dollar in total, and the machine will accept it and give you snack A. Similarly, you can choose to buy snack B and the machine will wait to receive 2 dollars. The ”accept states” are denoted with two circles in the graph.

However, the vending machine diagram is not exactly a finite automaton. Because we notice that the machine doesn't know where to go if the first input is not button A or B but some coins.

A set of strings that is accepted by the machine is called **the language of the machine**. In our vending machine example, the language is a set of money that is enough to buy a snack.



Very intuitively, you might already get an idea of what a finite state machine does. We can now look at the formal definition of a Deterministic Finite Automata.

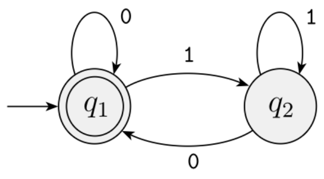
Definition 2.1. A deterministic finite automaton is a 5-tuple [1], where:

(1) Q is a finite set called the *states*,
(2) Σ is a finite set called the *alphabet*,
(3) $\delta : Q \times \Sigma \to Q$ is the *transition function*,

(4) $q_0 \in Q$ is the *start state*,
(5) $F \subseteq Q$ is the set of *accepting states*.

Here is a state diagram for the automaton M_1. We can see that there is an arrow pointing toward q_1 from nowhere, that indicates the *start state*; there are symbols like 0 and 1 on some arrows, these symbols make up the *alphabet*; Each circle is a *state* for the machine; Each arrow pointing from one state to another indicates a transition with an input at current state; The state with double circle is an accept state.

Example 2.1. *Consider the Finite Automaton M_1.*



We can describe M_1 formally by writing $M_1 = \{Q, \Sigma, \delta, q_1, F\}$

(1) $Q = \{q_1, q_2\}$
(2) $\Sigma = \{0, 1\}$
(3) We describe the transition function δ with a table:

State	**Input 0**	**Input 1**
q_1	q_1	q_2
q_2	q_1	q_2

(4) $q_1$is the start state
(5) $F = \{q_1\}$

The language of the machine M_1 is all the strings that ends with the symbol 0. In set notation :

$$L(M_1) = \{w0 \mid w \in \{0,1\}^*\}$$

In a Deterministic Finite Automaton (DFA), the transition function requires that for each state and input symbol, there is exactly one next state. However, in real-world scenarios like a vending machine, the system might have no response or multiple possible responses for the same input. To model such behavior, we introduce the concept of a Nondeterministic Finite Automaton (NFA), where the transition function can assign zero, one, or multiple possible next states for each state and input symbol pair.

Definition 2.2. A **nondeterministic finite automaton** is a 5-tuple [1]

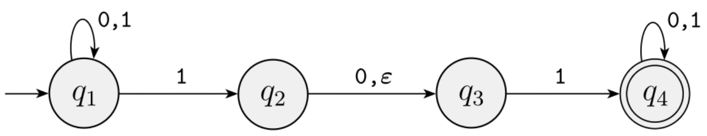
$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

(1) Q is a finite set called the *states*,
(2) Σ is a finite set called the *alphabet*,
(3) $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the *transition function*,
(4) $q_0 \in Q$ is the *start state*,
(5) $F \subseteq Q$ is the set of *accepting states*.

Σ_ϵ is the alphabet with an extra symbol ϵ which stands for empty string. And $\mathcal{P}(Q)$ is the power set of Q.

2.2. **Understanding nondeterminism.** From the definition above, it's not hard to see the difference between NFA and DFA. It mostly shows in the difference of transition function: the Σ_ϵ and $\mathcal{P}(Q)$. What feature does this transition function bring to NFA? Below is an example of M_2



Example 2.2.

When the machine is in the state q_1 and reads the input 1, here are two options: stay in q_1 or go to q_2. Similarly, when the machine is in state q_2, it can go straight to q_3 since there is an ϵ (empty string, exists between any two symbols) or read 0 and go to q_3 (if the next input is 0). However, it does not have the next destination if the next input is 0 and the machine is at state q_3.
In order to get to the accept state q_4, we need to have the string 101 or 11 in our input. Thus the language of the machine M_2 is any string that contains a substring 101 or 11. In set notation

$$L(M_2) = \{xyz \mid x, z \in \{0,1\}^*,\ y \in \{101, 11\}\}$$

Another problem the output $\mathcal{P}(Q)$ brings us is that $\emptyset \in \mathcal{P}(Q)$, so there might be no next state for the machine to go to, and the process "dies" or "halts" there. For an NFA, it is not certain that the machine will go to the next state from the current state because 1. There are multiple states to go to 2.and there is an empty string that is accepted as a reading. With this feature, the machine goes to multiple states, which are multiple branches. When the NFA is working, if it chooses one branch and continues in that specific branch, if the result does not bring the machine to an accept state, the machine cannot immediately decide to reject the input. The reason is that the machine needs to backtrack to the other branches until every branch is rejected. To understand how this works better, we are introducing Depth-First Search, which is a method in graph traversal that has a very similar process. Before that let's take a look at graph.

2.3. **Graph.**

Definition 2.3. An **undirected graph** is an ordered pair G=(V,E) where V is a set of vertices, and E $\subseteq \{\{x, y\} \mid x, y \in V$ and $x \neq y\}$.

Definition 2.4. A **tree** is an undirected graph which any two vetices are connected by exactly one path.

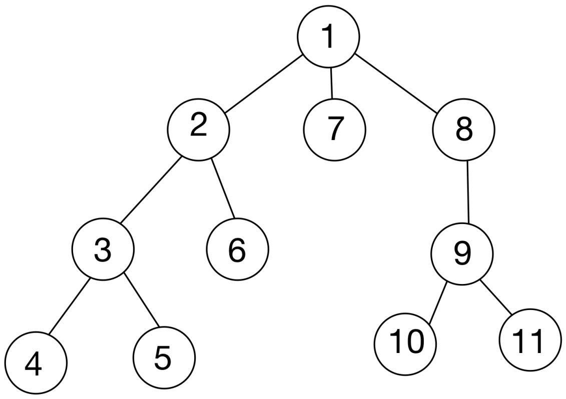


FIGURE 2. Tree search

Definition 2.5. A **neighbor** of a node is any other node that is directly connected with it.

Definition 2.6. A **stack** is an abstract data type that serves elements in an order of first in last out.

2.4. **Depth-First Search.** In order to understand nondeterminism in computation, it is helpful to first consider how a deterministic algorithm explores a graph. One way is to perform a depth-first search (DFS). DFS systematically explores a graph by starting at a source node and searching as deep as it could before backtracking. Whenever we encounter a node with multiple outgoing paths, we choose one path to follow and store the remaining adjacent nodes on a stack. Once we reach the end of that path or a dead end, we backtrack by popping from the stack and continue the search. The following is an example of a graph and how we search it. The number on each node is also the order of search. First, we start from source node 1. There are three outgoing paths, 2, 7, and 8. We put 8,7,2 into stack in order. The top one is 2 so that is the node we are going to search next. Take out 2 from the stack and put in 6,3. The next top item in the stack is then 3. Take 3 out of the stack and put in 4, 5 into the stack. After we searched both 4 and 5, and take both of them out of the stack, the top item in the stack is 6, and we backtrack to 6 and so on.

Example 2.3.

Let us look at our previous NFA in Example 2.2. Although the the idea of nondeterminism is a strong theoretical tool that allows us to compute things with presenting parallel explorations, the real-world computation remains deterministic: the NFA read the input symbol and see what pathways it could take according to the transition function, and then do the same process as we described in DFS. In the Tree diagram below, we show the process of state search for the machine M_2 with in put string 01011. With the first input 0 started at q_1 the only accessible next state is q_1. Similar to the idea of a DFS for graphs where we push the neighbors into the stack, we push q_1 into the stack. Second, we search the top element in the stack which is now q_1, we pop q_1 out and then push in the three "neighbors" with the input 1, which are q_1, q_2 and q_3. The order of pushing

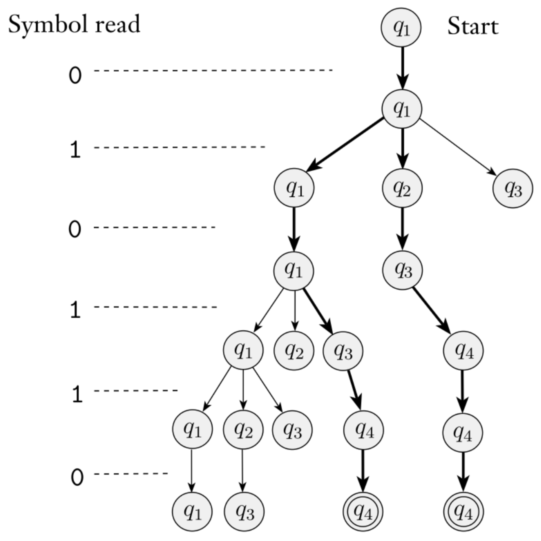
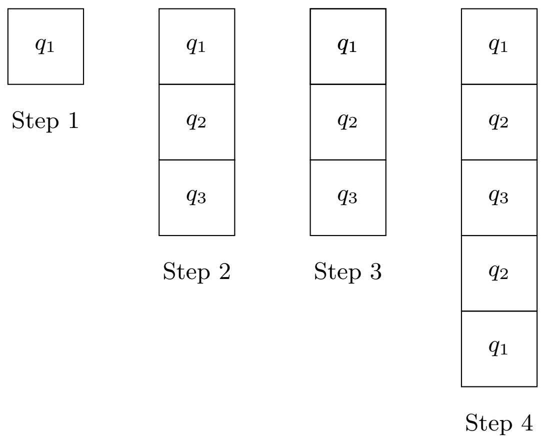


FIGURE 3. DFS of M_2 [1]

in the accessible states does not influence the whole process of search and result. The third step is fundamentally the same as the first step, we pop out the o state that we are searching, and push in the only "neighbor" with input 0 at state q_1. And again, the fourth step is the similar to the second step, we pop out q_1 and then push in q_1, q_2 and q_3. The path ends when the input has been fully consumed. By using a stack, we can backtrack to the most recent branching point. Once an accepting state is reached, the search can terminate, as nondeterminism guarantees acceptance if at least one computation path leads to success.



3. Turing Machine

To understand the limits of computation, we need a more powerful model than Automaton – Turing Machines (TMs). A Turing Machine is a powerful machine with an infinitely long tape that we can write and read from. This allows Turing Machines to go beyond the limit of Finite Automatons. It is the foundation of modern computer science.

Definition 3.1. A Deterministic Turing machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, 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 $\llcorner$,
(3) Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
(4) $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
(5) $q_0 \in Q$ is the start state,
(6) $q_{\text{accept}} \in Q$ is the accept state, and
(7) $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

The $\{L, R\}$ part tells whether to move the tape head left or right (L or R).

Definition 3.2. The set of strings that a TM accept is called **the language of the TM**, denoted by L(TM).

For Turing Machines, there are three outcomes: accept, reject, or loop. Here we have some new definitions due to this feature.

Definition 3.3. Call a language **Turing-recognizable** if there exists some TM recognize it.

Definition 3.4. Call a language **Turing-decidable** if there exists some TM accept or reject it. These machine are called **deciders** because they always decide to accept or reject a string.

Now we construct a Turing machine M over the alphabet $\{0, 1\}$ that accepts strings with the same number of 0s and 1s.

Example 3.1. *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ be a Turing Machine where:*

(1) $Q = \{q_0, q_1, q_2, q_3, q_{accept}, q_{reject}\}$
(2) $\Sigma = \{0, 1\}$
(3) $\Gamma = \{0, 1, X, Y, \sqcup\}$
(4) q_0 *is the start state*
(5) q_{accept} *is the accept state*
(6) q_{reject} *is the reject state*
(7) δ *is defined as follows:*

Current State	*Read Symbol*	*Action (Write, Move, Next State)*
q_0	0	(X, R, q_1)
q_0	1	(Y, R, q_2)
q_0	X	(X, R, q_0)
q_0	Y	(Y, R, q_0)
q_0	$\sqcup$	$(\sqcup, R, q_{accept})$
q_1	0	$(0, R, q_1)$
q_1	X	(X, R, q_1)
q_1	Y	(Y, R, q_1)
q_1	1	(Y, L, q_3)
q_1	$\sqcup$	$(\sqcup, R, q_{reject})$
q_2	1	$(1, R, q_2)$
q_2	X	(X, R, q_2)
q_2	Y	(Y, R, q_2)
q_2	0	(X, L, q_3)
q_2	$\sqcup$	$(\sqcup, R, q_{reject})$
q_3	0	$(0, L, q_3)$
q_3	1	$(1, L, q_3)$
q_3	X	(X, L, q_3)
q_3	Y	(Y, L, q_3)
q_3	$\sqcup$	$(\sqcup, R, q_0)$

Example 3.2.

Now let's examine the input 01 on M. Starting on input `01` at state q_0 with the head on the first symbol, the machine marks the first 0 as X and moves right to state q_1; then it finds a 1, marks it as Y, moves left to state q_3 to return the head to the left end; finally, it scans right back to the blank and transitions to the accept state.

Just like Finite Automatons, we also have a variant of TM that is Nondeterministic Turing machines. The definition is similar to the Deterministic Turing Machine, the only difference is the transition function which shows its nondeterminism. It has the form:

$$\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$$

In order to know at what stage our computation has been going so far, we need to know three things: What state are we at, what is the content on the tape, and where the head is pointing at.

We call these three things **configuration**.

$$\begin{gathered} q_0\ 01 \\ \mathrm{X}\ q_1\ 1 \\ q_3\ \mathrm{X\ Y} \\ \sqcup\ q_3\ \mathrm{X\ Y} \\ q_0\ \mathrm{X\ Y} \\ \mathrm{X}\ q_0\ \mathrm{Y} \\ \mathrm{X\ Y}\ q_0\ \sqcup \\ \mathrm{X\ Y}\ \sqcup\ q_{\mathrm{accept}}\ \sqcup \end{gathered}$$

Example 3.3.

A feature we noticed is that we cannot do the same deterministic process of DFS as for NFA for NTM, because a NTM might end up in a loop and never end up in an accept state. We define a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}})$ as follows:

(1) $Q = \{q_0, q_1, q_{\text{accept}}\}$
(2) $\Sigma = \{1\}$
(3) $\Gamma = \{1, \sqcup\}$
(4) q_0 is the start state
(5) q_{accept} is the accept state
(6) The transition function δ is given by:

Current State	Read Symbol	Write Symbol	Move	Next State
q_0	1	1	R	q_1
q_1	1	1	L	q_0
q_0	$\sqcup$	$\sqcup$	R	q_{accept}

On input 1, the machine loops infinitely by alternating between states q_0 and q_1, moving the head right and then left without ever reaching the blank symbol. Hence, it never halts.

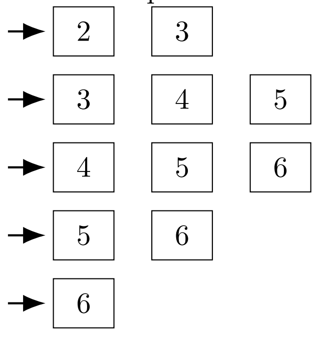
So, here we introduce another type of searching method called Breadth-First Search(BFS).

Definition 3.5. A queue is an abstract data type that serves elements in an order of first in first out.

3.1. **Breadth-First Search.** For a specific input for TM, we can have a **Computation Tree** which all the nodes of it are different configurations. Since the TM can potentially end up in a loop and never halt, we cannot do DFS. Because using DFS may lead us into an infinitely deep path before backtracking. Instead, we visit all the configurations in breath-first search to make sure that if there exist an accept state, we will visit it at some point.

Example 3.4.

In order to do Breath-First Search, we start with one and put all of its neighbors into a queue and search from the beginning of the queue. In the example, we first visit 1, and put 2 and 3 into the queue. Now the first element in the queue is 2, so we put neighbors or 2 into the queue which is 4 and 5 and take 2 out of the queue. The next element in the queue is 3, so we put 6 into the queue and take 3 out and so on. Good thing about BFS is that if there exist any accept states, we would not skip it or fail to find it because of potential loops.



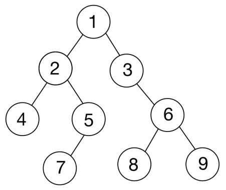


FIGURE 4. Breadth-First Search example

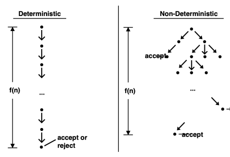


FIGURE 5. TM searching [1]

CONCLUSION

Both Depth-First Search (DFS) and Breadth-First Search (BFS) are fundamental algorithms that systematically explore the shortest paths in graphs and ensure that all nodes are visited. When it comes to nondeterminism, these two searching strategies help us simulate a nondeterministic process in a deterministic way. Both of these two strategies bridge the gap between theoretical nondeterminism and practical process, making them useful tools in computer science.

REFERENCES

[1] Sipser, Michael. *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2012.
[2] Russ Cox. *Regular Expression Matching Can Be Simple and Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)*. January 2007.