Automata Theory Notes

Introduction

In this course we will look at: - Which problems are computable? - Design abstract models of computation and try to understand what problems can be solved by them. - What are the limits of computational models?

Consider a simple robot which has a power button, and can move either forward or backward when on. It has a sensor that recognizes if there is an obstacle in front of or behind it and it moves accordingly away from any nearby obstacle.

This represents a system which can be described as such in terms of its states and inputs

- States: {OFF, FORWARD, BACKWARD}
- Input: {BUTTON, SENSOR}
- Initial state: OFF

The robot can transition from one state to another using the input signal.

We can draw a state transition table, and a state diagram for the same

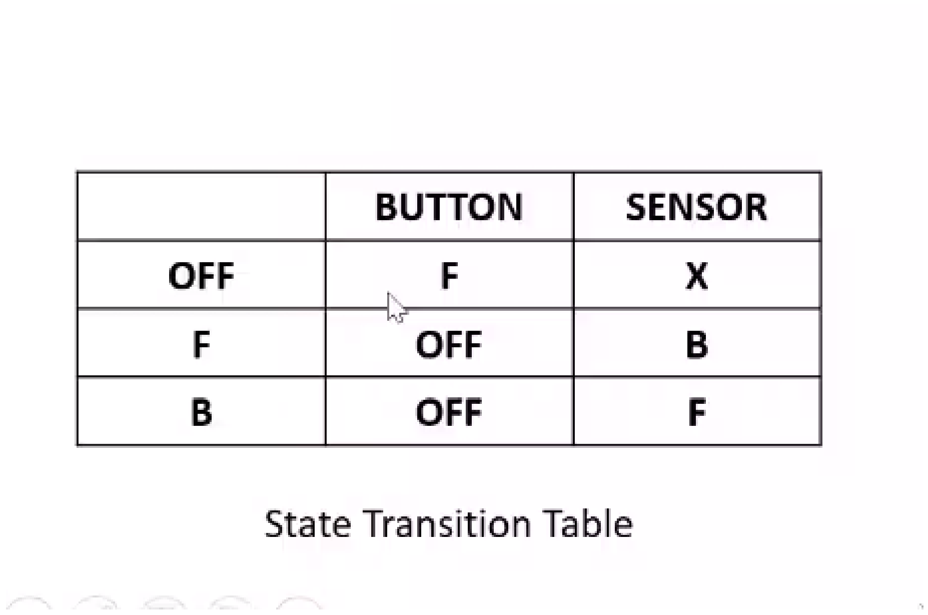
We care about whether a given problem can be computed by a particular computational model. What does this mean?

Suppose the problem is to sort a list of numbers. This is a general problem, and we may have a specific instance of it that we wish to solve, for e.g. we might wish to sort the list `[2, 3, 5, 1, 7, 10]`.

Now every general problem, can be converted into a decision problem! We have two sets, a YES set containing all the instances where the answer is YES, and a NO set containing all the instances where the answer is NO.

Consider a computational model, and a problem P. If for all inputs belonging to the YES instance set of P, the device outputs YES, and for all inputs belonging to the NO instance set of P, the device outputs NO, then we can say that the problem is **computable** by this computational model

Can we have problems that cannot be solved by ANY computer? Halting problem as described by Turing.



	BUTTON	SENSOR
OFF	F	X
F	OFF	B
B	OFF	F

State Transition Table

Figure 1: State Transition Table

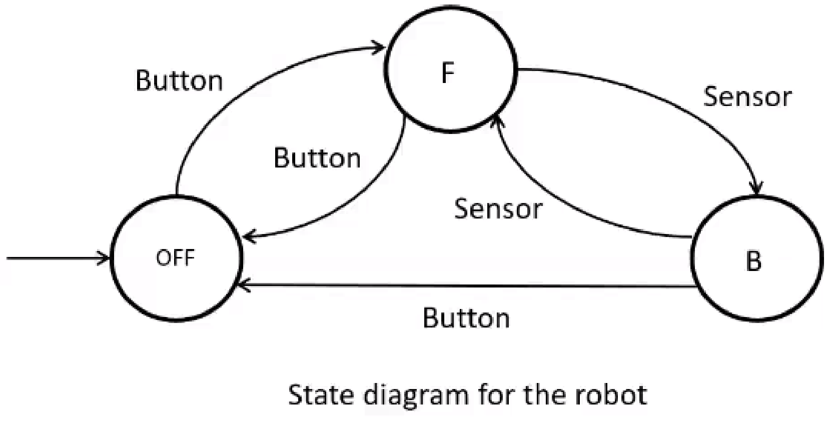


Figure 2: State Diagram

Consider an algorithm M, the master algorithm, It takes two inputs, another algorithm, and some input for the input algorithm. Let us call the input algorithm C, and the input for C as X.

The master algorithm outputs YES if C terminates on the input X, and outputs NO if C does not terminate on input X.

Unfortunately, such an algorithm cannot exist. Why?

Let us consider another, more grounded example. "Does a polynomial P(x, y) with integral coefficients with integral coefficients have integral roots?"

Some Terminology

Alphabet Any finite, non-empty set of symbols, often represented with the symbol Σ

Strings/Words Finite sequence of symbols from an alphabet. We often use ϵ to represent the empty string.

Language Set of words/strings from the current alphabet

Models of Computation

Deterministic Finite Automata (DFA) Model

It is deterministic in the sense that from a given state, upon accepting a certain input, it can only perform a single deterministic transition.

Formally, a DFA can be represented as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q represents the finite set of states.
- Σ is the input alphabet.
- $\delta : Q \times \to Q$ is the transition function.
- q_0 is the initial state.
- $F \subseteq Q$ is the set of final/accepting states.

Its characteristics are:

- Single start state
- Unique transitions
- Zero or more final states

Constructing a DFA for a language

Example: $\Sigma = \{0, 1\}$, $L(M) = \{\omega \mid \omega \text{ is divisible by } 3\}$

- The DFA will have three states, each corresponding to one of the three possible remainders, 0, 1, 2.
- The final state is where remainder is 0.

Sometimes it might be easier to construct a toggled DFA, solving for $\overline{L}$ instead of L.

Non-deterministic Finite Automata (NFA)

Non-deterministic finite automata can take undergo more than one possible state transition for given a certain input.

- Q represents the finite set of states.
- Σ is the input alphabet.
- $\delta : Q \times \to 2^Q$ is the transition function.
- q_0 is the initial state.
- $F \subseteq Q$ is the set of final/accepting states.

Note that since the transition can happen to more than one state from the current state, the range of the transition function is the set of all subsets of Q, i.e. the powerset of Q, 2^Q

Characteristics of NFA:

- Single start state
- Zero or more final states
- Multiple transitions are possible on the same input for a sate
- Some transitions might be missing
- ϵ-transitions

Are NFAs more powerful than DFAs?

NFA vs DFA

As it turns out, the power of NFAs and DFAs is completely equivalent! The languages that they can handle are equivalent. We can show this by converting an NFA to a DFA.

In order to do this, we can construct a "remembering DFA". This is done in two steps:

1. Make a table of transitions on possible inputs for all the states. This includes the input of an empty string (ϵ-transition).

2. Using this table, make a DFA, where each state in the DFA represents the set of all states that could be reached in the NFA. For example, if the initial state q_0 has an epsilon transition to the state q_1, then the initial state in our DFA should be $\{q_0, q_1\}$, because this represents the set of all states we could be at "initially" (having taken no input so far).

A language is called a **regular language** is there exists some finite automata that can identify it.

Regular operations:

- Union
- Concatenation
- Star

Regular languages are closed with respect to all of these operations.

Suppose L_1 and L_2 are regular languages. Then their union $L_1 \cup L_2$ must also be a regular language.

If we have an NFA for L_1 and one for L_2, then we can make an NFA for their union simply by adding a state behind the both of them, which adds an epsilon transition to the first NFA, and another to the second NFA.

Similarly, the concatenation of the two languages, $L_1 \cdot L_2$ must also be a regular language.

We can construct an automata for the resultant language by taking the DFAs for L_1 and L_2, and taking all the accepting states of L_1, and adding an epsilon transition from each of these to the initial state of L_2.

Is the set of all regular languages closed under complement?

Given a DFA M, such that $L(M) = L$, construct the toggled DFA $M\prime$ from M, by changing all the non-final states to the final states and vice versa. This new DFA will accept $\overline{L}$.

Regular Expressions

Regular expressions describe regular languages algebraically.

Syntax for regular expressions:

- Φ is a regular expression, $L(\Phi) = \Phi$
- ϵ is a regular expression, $L(\epsilon) = \{\epsilon\}$

DFA to Regular Expressions: GNFA

- GNFAs may have regex arrows

In order to convert a DFA to a regular expression, we convert into a GNFA. Then we can remove one state in the GNFA, and replace it with arrows labelled with an appropriate regular expression. Thus we end up with a GNFA with one less state.

We can repeat this as many times as needed, till we have only two states, the start state, and the end state, and the arrow between them is labelled with our desired regular expression.

Pumping Lemma

So far we have shown that the following statements are equivalent

- L is a regular language
- There is a DFA D such that $L(D) = L$
- There is an NFA N such that $L(N) = L$

- There is a regular expression R such that $L(R) = L$

However, not all languages are regular.

Let us take an example, $\Sigma = \{0, 1\}$. Consider the language $L = \{0^n 1^n | n \geq 0\}$. If we say we have a DFA for L, with n states (say $n = 10$), then $0^{10}1^{10}$ must be accepted.

By the **pigeonhole principle**, while reading the first ($n = 10$) symbols, How many some states must be revisited. Hence some loop must be present. Let us say that ($t = 3$) states are present, in this loop on a 0 state. If the DFA accepts $0^n 1^n$, it must also accept $0^{n+1}1^n$, because we can just take the loop an extra time.

This poses a contradiction, as $0^{n+t}1^n$ should not be accepted by the DFA. Thus, the language L is **not** regular.

Lemma Let L be a regular langauge, then there exists an integer $p \geq 1$ depending only on L such that for every string w in L of length at least p (known as the pumping length), can be written as $w = xyz$ such that:

- $|y| > 0$
- $|xy| \leq p$
- $\forall i \geq 0, \quad xy^i z \ inL$

Proof Sketch Given a DFA M of p states, in any run in it corresponding to strings of length $\geq p$, some states are repeated.

Suppose $s = s_1 s_2 ... s_n$, is such a string, and $r_1 r_2 ... r_{n+1}$ is the sequence of states encountered in the run.

As $n + 1 \geq p + 1$, at least two states must be repeated, let us call them r_j and r_l, i.e. $r_j = r_l$, but $r \neq l$.

So we can divide the string into three parts:

- $x = s_1 ... s_{j-1}$, which takes us from r_1 to r_j.
- $y = s_j ... sd_{l-1}$, which is the loop.
- $z = s_l ... s_n$, which brings us to the final state.

Now, we can traverse the loop bit any number of times, and the string we get as a result belongs to the language. In a regular language, this is always true.

Thus, this provides us with a reliable way to check if the language is regular. A regular language which accepts strings of length greater than or equal to the number of states in a DFA for it, must necessarily be able to pump through some loop like this.

Grammars

Grammars provide ways to generate strings belonging to a language. For some classes of grammars, one can build automata that recognizes the language generated by the grammar.

If the grammar is of the form

$$\begin{aligned} Var &\to TerVar \\ Var &\to Ter \\ Var &\to \epsilon \end{aligned}$$

then the language of the grammar is regular, and it is known as a right-linear grammar (all variables are to the right of terminals in the RHS)

If the grammar is of the form

$$\begin{aligned} Var &\to VarTer \\ Var &\to Ter \\ Var &\to \epsilon \end{aligned}$$

then the language of the grammar is regular, and it is known as a left-linear grammar (all variables are to the left of terminals in the RHS)

Note that mixing left-linear grammars and right-linear grammars won't generate regular languages.

L-systems

Originally created by Lindenmayer to model bacteria growth patterns

L-systems are a type of a formal grammar and a parallel rewriting system.

L-system is a formal system, It is defined as a 3-tuple $G = (V, w, P)$, where V is the alphabet set, w is the start/initiator or axioms, and P is the set of production rules.

Production Rules

F move forward by d steps, + is rotating towards one direction by angle δ, and - is rotating towards the opposite direction by angle δ, `[]` indicate branching

We could also have stochastic L-systems, where we have probability for certain branches to be taken.

Context-free Grammars

Languages with production rules of the form:

$$V \to (V \cup T)$$

Regular languages are a subset of context-free languages.

For example, if we consider the grammar G with the production rule:

$$S \to 0S1|\epsilon$$

This would produce strings of the general form 0^n1^n, which we had seen earlier is **not** a regular language, but it is a context-free language

CFL might be unions of simpler languages, which can be a useful thing to keep in mind

Parsing and Ambiguity

Leftmost derivations, rightmost derivations and general derivations. If a string can be parsed in more than way within a grammar, then ambiguity exists in the grammar

Chomsky Normal Form

A CFG G is in CNF if every rule of G is of the form

$$\begin{aligned} Var &\to VarVar \\ Var &\to ter \\ StartVar &\to \epsilon \end{aligned}$$

If $w \in L(G)$, then a CFG in CNF has derivations of $2n-1$ steps for input strings w of length n. Thus, given a string w, where $|w| = n$, we can check whether the grammar can generate w or not by performing all derivations of $2n-1$ steps.

Theorem A CFG in Chomsky Normal Form has derivations of $2n-1$ steps for generating strings $w \in L(G)$ of length n.

Proof Note that any CFG in CNF can be written as:

$$\begin{aligned} &A \to BC && [B, C \text{ are not start variables}] \\ &A \to a && [a \text{ is a terminal}] \\ &S \to \epsilon && [S \text{ is the start variable}] \end{aligned}$$

Basis step: Let $|w| = 1$. Then one application would suffice, $2|w| - 1 = 1$ step.

Induction step: Let the statement hold for $|w| = k$. Assuming $|w| > 1$, any derivation will start from $A \to BC$. So $w = xy$, where $B \implies x, |x| > 0$ and $C \implies y, |y| > 0$.

The number of steps needed to derive w will be the number of steps needed to derive x and y, and the step to combine them, which is

$$(2|x| - 1) + (2|y| - 1) + 1 = 2(|x| + |y| - 1) = 2|w| - 1$$

Converting a CFG to CNF

CYK Algorithm

The CYK algorithm is for parsing a grammar in CNF. It runs in polynomial time, as opposed to the direct method with CNF which takes exponential time.

Pushdown Automata

Any automata that recognizes all context free languages will need unbounded memory. For this we can utilize pushdown automata, which has an infinite stack and an infinite one way input tape. It is a common convention to signify that a stack is empty by saying the top element is a symbol $.

When representing transitions in a PDA, we need to mention three things in it, the input symbol, the value for the stack top, and the final operation to perform in that case.

Note that in order to read the stack top, we have to pop the top value off the stack.

a, b, Pop however means just a single pop, just to be explicit. a, b, ϵ would be equivalent.

The notation used by some other references, such as Sipser, is $a, b \to c$ which means input tape has a, stack top has b, and the operation is to push c. $a, b \to \epsilon$ would then be just a pop if the top is b.

Formally, a finite automaton M is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q represents the finite set of states.
- Σ is the input alphabet.
- Γ is the stack alphabet.
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accepting states.

PDA to accept palindromes

We can make such a PDA by keeping in mind the following things:

- We want to push the first half onto the stack, and then pop symbols off the stack if they match the second half of the input.
- How will the PDA know when we're at the middle? It won't. Simply add an epsilon transition from the first part (where you're pushing symbols on) to the second part (where you're popping symbols off). We will end up in a crash state everytime we make that epsilon transition and we're NOT in the middle of the string, but in just one of the transitions we reach the accepting state if it is indeed a palindrome, which is enough.

- This only works for even length palins though. For odd length palins, we need to account for the unmatched middle character. How do we do this? Non-determinism again. We add a, ϵ, ϵ transitions between the first part and the second part, where a is symbol from the input alphabet.

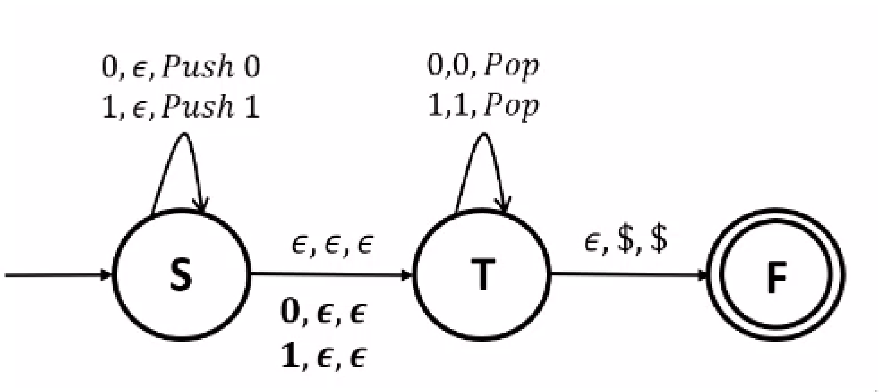


Figure 3: diagram for binary strings