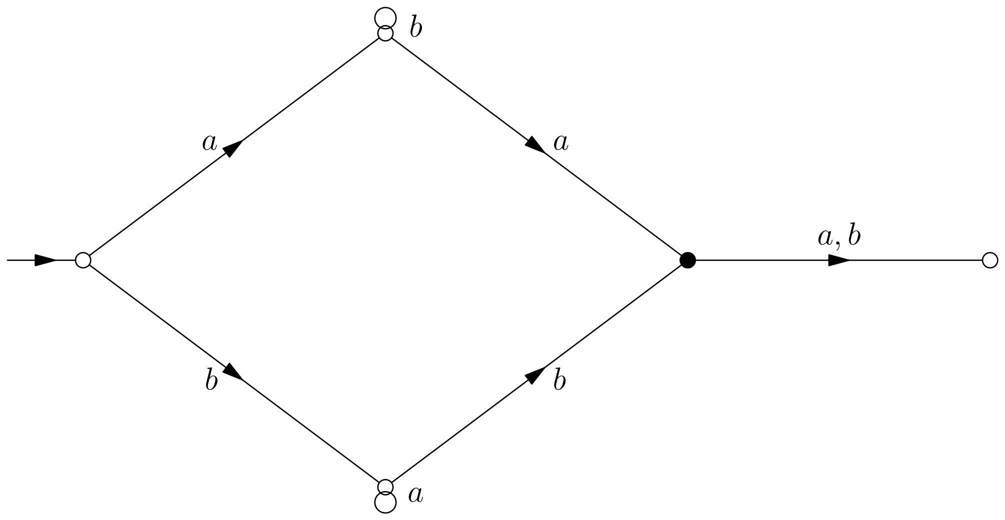
THE LIMITS OF COMPUTATION

JEREMY BOOHER

What are the limits of computation? Can a sufficiently powerful computer, or sufficiently clever mathematician, solve any problem? We will investigate this problem, first in a very limited setting and then in a very general computational model. In both cases, there will be natural problems that cannot be solved.

1. Finite State Automata



This diagram describes a very primitive method of computation called a finite state automata. It decides whether to accept or reject a string composed of the letters a and b. For example, given the string baab, it computes as follows: it starts at the left node, with the arrow from nowhere. The first letter is b, so it follows the arrow labeled b down and right. The next letter is a, so it follows the loop and stays at the same node. Likewise for the next a. Next comes the letter b, so it follows the arrow up and right to the filled in circle. There are no more letters, so we end at the filled in circle. Because we ended at that circle, the machine accepts the string baab.

On the string b, the machine heads down and right, and stops there. The circle is not filled in, so the machine rejects the string b.

The nodes of the graph are called states, and the arrows are transitions. So the machine changes state according to the current state and the next letter of the input. If it ends in an accept state (marked with a filled in circle), it accepts the string. Otherwise it rejects it.

Exercise 1. *Check that this machine accepts the string aa but rejects aab.*

The set of all strings (composed of a and b) that this machine accepts is called the language it accepts (or the language it decides). Here are some ways to write down languages:

- Strings of letters. a^n is shorthand for n copies of a.
- Unions: $a \cup b$ is the language containing a and b.
- Concatenation: two sets of strings, one followed by the other. $(a \cup b)b = \{ab, bb\}$.
- a^*, which is the set of zero or more copies of a. This operation is called the Kleene star.

Date: July 18, 2011.

Exercise 2. *Describe the language this machine accepts.*

Finite state automata can describe lots of languages.

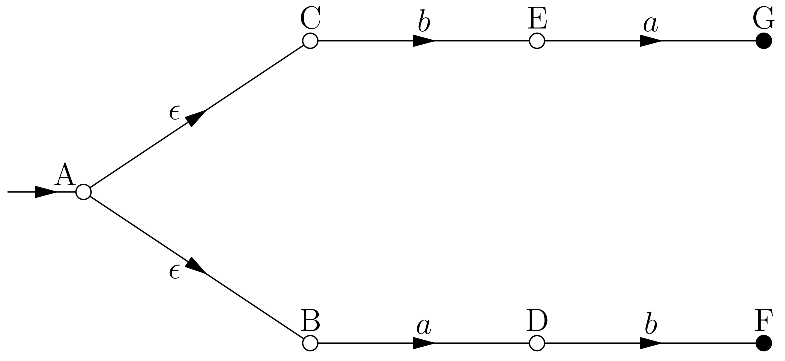
Exercise 3. *Which of the following can be decided by a finite state automata?*

(1) aab
(2) 0^*1^*
(3) $\{0^n1^n : n \in \mathbb{N}\}$
(4) 0010^*1^*
(5) $ab \cup ba$

If you try to write down finite automata that accept these languages, we encounter several annoying features. First, we usually include one (or more) states which are not accept states and which it is impossible to leave, signifying that the computation has failed. The convention is to not draw these states, and if for a given state and input there is no outgoing arrow for that input symbol, the computation will fail.

Secondly, we allow non-determinism. This means that there can be multiple arrows labeled with the same letter leading out of a vertex. When faced with that input, the finite automata makes all choices, and continued the computation for each one. If any computational path accepts the input, the string is accepted. If none do, it is rejected. Non-determinism also allows the possibility of ϵ arrows: these change the state without reading any symbols of the string. This is very helpful in describing automata succinctly.

Exercise 4. *Check that the following automata accepts the language $ab \cup ba$.*

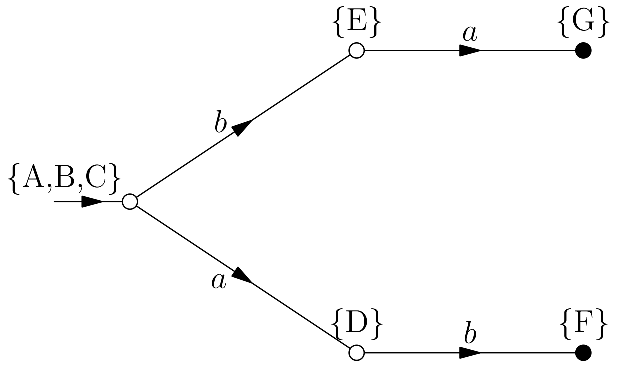


Despite the added complexity of a non-deterministic finite automata, they actually aren't any more powerful.

Theorem 5. *Non-deterministic finite automata can decide a language if and only if deterministic finite automata can decide the language.*

Proof. Any deterministic machine is also a non-deterministic one, so all we need to show is that a deterministic machine can simulate a non-deterministic one. This construction is called the power-set construction. Let A be a non-deterministic finite automata. Take the power set of all states of A to be the states of the new machine. Define the transition arrows as follows: given a collection of states of A, under the input a the arrows goes to the collection of all states of A reachable with input a: in other words, follow some number of ϵ arrows (possibly zero) then follow an arrow labeled a. The states reachable starting at any state in the collection are the ones in the new collection. This defines the arrows on the power set. The start state is the start state of A. The accept states are all those collections of states where there is a path of ϵ arrows starting at one of the states and ending at an accept state of A (a path of zero ϵ arrows is allowed). This machine accepts exactly the same strings as A does, but is non-deterministic. It is of course substantially more complicated. □

If we apply the power set construction to the non-deterministic automata accepting $ab \cup ba$, the interesting part of the resulting automata is pictured.



The class of regular languages are those languages that can be written down in terms of the letters of the alphabet, unions, concatenation, and the Kleene star. For example, all of the languages in Exercise 3 are regular, except possibly $\{0^n 1^n : n \in \mathbb{N}\}$.

Theorem 6. *Every regular language can be decided by a finite automata.*

Exercise 7. *Prove this by showing how to take an automata for a regular language and construct a larger automata that recognizes the Kleene star. Do the same for unions, concatenation, and single letters. For example, the construction for unions is illustrated above in the case $ab \cup ba$.*

It is in fact true that the set of strings a finite automata recognizes is always a regular language. This is somewhat less intuitive. For details, see Sipser [1].

2. The Limits of Finite State Automata

It should come as no surprise that finite state automata are quite limited. After all, they have no memory, and can only look at their input in order. Regular languages, the class of languages they can decide, are quite limited. So how would one show that **no** finite state automata can decide a language? The answer is a result called the Pumping Lemma.

Theorem 8 (Pumping Lemma)**.** *For any finite automata, there exists an integer p such that for any string w of length at least p accepted by the automata, w can be split up into three strings x, y, z such that $w = xyz$ and*

- $|y| > 0$.
- *$xy^i z$ is accepted for all $i \geq 1$.*
- $|xy| \leq p$.

Proof. View the deterministic finite automata as determining a graph, and any input string as determining a path in this graph. Let p be the number of states. Then any path of length p (coming from a string of length p or more) must pass through $p+1$ states. By the pigeonhole principle, some state s occurs twice. Let x be the string that gets from the start state to s, y the string that gets from s back to s for the first time, and z the string that gets from s to an accept state. It is clear that $|y| > 0$. Then $|xy| \leq p$ since the first repetition of states must occur within the first p transitions. $xy^i z$ is accepted because repeating the string y simply repeats the loop from s back to s in the automata. $\square$

The pumping lemma can easily show that certain languages are not regular.

Example 9. Consider the language $\{0^n 1^n : n \in \mathbb{N}\}$ consisting of equal numbers of 0s and 1s. Suppose a deterministic finite automata accepted this language. Then there would be a pumping length p such that any string of length at least p can be divided up as in the pumping lemma. In particular,

$0^p 1^p = xyz$. Since $|xy| \leq p$, y contains only 0s. As $|y| > 0$, y contains at least one zero. As xyz contains an equal number of 0s and 1s, xy^2z contains more 0s than 1s. But the pumping lemma guaranteed that xy^2z is accepted by the automata. Therefore no finite automata can accept this language. In a sense, this shows that finite automata cannot count.

Example 10. As a more complicated example, we will also show that no finite automata can check whether a number is a perfect square. Consider the language $\{1^{n^2} : n \in \mathbb{N}\}$. Suppose a deterministic finite automata accepted this language. Then there would be a pumping length p, so we can split $1^{p^2} = xyz$ subject to the constraints in the pumping lemma. Let r, s and t be the lengths of x, y and z. Then the condition that the machine accept xy^iz means that for any $i \geq 1$, we have $r+t+si$ is a perfect square. However, $s \neq 0$ as $|y| > 0$, and no arithmetic progression can consist of only perfect squares (exercise: prove this).

3. The Limits of Computation

Now that we have understood this very simple method of computation, we will now turn to the most powerful models of computation. The formal model studied in computer science is called a Turing machine, but experience has shown it is the same as what a physical computer can do or what a human brain can compute. This (unprovable) statement is called the Church-Turing thesis. For this reason, we will not worry about the exact model of computation. The key to finding the limits of computation lies in the ability to represent programs by numbers. This can be done formally with Turing machines, but it is most obvious on a physical computer: all programs are stored in memory as a sequence of bits. These bits determine a number written in binary.

Suppose we had a program HALTING that could solve the problem "The nth program halts on input s". By this, we mean that the program HALTING takes two inputs, n and s, and returns true if program n halts when run on input s, false otherwise. Construct a new program P that on input n does the following: run HALTING on input (n, n). (This checks if program n halts on input n.) If it returns true, enter an infinite loop. Otherwise return true.

Let m be the number corresponding to program P. What does program P do on input m? Suppose P halts on input m. Then tracing through the description of P, P runs the program HALTING on program m with input m. However, program m is P itself. We assumed it halts, so HALTING returns true. But then the instruction in P is to enter an infinite loop. Therefore P does not halt. Suppose P does not halt on input m. Then tracing the description, the second case occurs so P returns true. Again, a contradiction. Therefore in either case we have a contradiction. This implies the program HALTING cannot exist. In other words, no computer program (Turing machine, algorithm, or human) can tell in general whether a given program halts on a specified input.

This is less surprising than it may sound. Consider a computer program that writes down all strings of symbols and checks whether they constitute a valid proof of the Riemann hypothesis. (There are formal languages where proofs are written down very rigorously and can be verified by computer programs. Assume we use one of these.) If the proof is valid, the program halts. If the proof is not valid, continue. If the Riemann hypothesis is provable, this program eventually halts. If it is unprovable, the program never halts. Thus a decider for the halting problem would allow all mathematical problems to be trivialized.

4. Further Reading

Sipser's book [1] is an excellent reference on the theory of computability and complexity. It is written for an undergraduate audience, and contains numerous good exercises.

References

1. Michael Sipser, *Introduction to the theory of computation*, Thompson Course Technology, 2006.