CDM
Memoryless Machines

Klaus Sutner
Carnegie Mellon Universality

10-memoryless 2017/12/15 23:19

CC BY NC SA

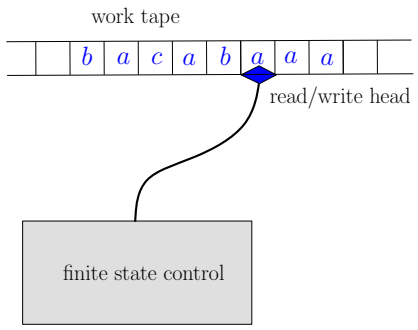
Where Are We? 3

We have a description of abstract computability in terms of Turing machines, and various equivalent models. Abstract here refers to the fact that we totally ignore resource constraints, such as time, space and energy, that are critical in actual computation.

Primitive recursive functions are more restricted, but they, too, are much too powerful for practical computation.

To get more realistic models, we first plunge to the very bottom: machines without memory. Surprisingly, they still have lots of interesting properties, and are very useful in practical applications.

Turing Machines 4



Very Concrete Computability 5

So far we have seen a description of abstract computability in terms of register machines and general recursive functions. Abstract here refers to the fact that we totally ignore resource constraints (time and space).

A more restricted class of computable functions are the primitive recursive ones – but they, too, are much too powerful for practical computation. Kalmár's elementary functions are more realistic.

In modern complexity theory, the class $\mathbb{P}$ of polynomial time computable (decision) problem are generally considered to be an excellent match with the intuitive notion of efficient computation.

Here is a model that pushes the resource restrictions to the very limit – but still has many surprising and important applications.

Taming Turing Machines 6

The central problem with general Turing machines is that we have no way of predicting the amount of tape used during a computation (which could be used to also obtain a bound on the length of the computation).

So how about simply imposing a bound on the amount of tape that the machine may use? If the machine attempts to use more tape, the computation simply fails.

A fairly natural restriction would be to allow only as much tape as the input takes up originally: think of two special end markers

$$\#x_1x_2\ldots x_{n-1}x_n\#$$

where the head is originally positioned at the first symbol of x. The head is not allowed to move beyond the two cells marked $\#$.

Linear Bounded Automata

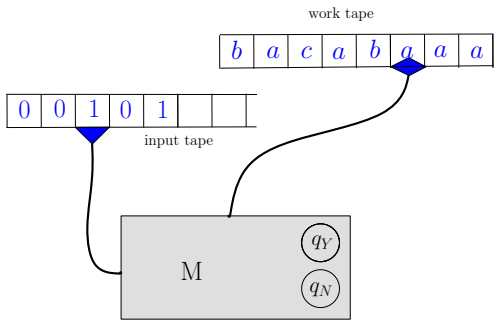
This leads to an important class of machines: linear bounded automata (LBA) and the corresponding class $\mathrm{SPACE}(n)$ of problems solvable in linear space; introduced in their deterministic form in 1960 by Myhill, and in their full nondeterministic form in 1964 by Kuroda.

Unfortunately, LBAs are still much too powerful and complicated. E.g., nondeterministic LBAs can accept every context-sensitive language.

Ominously, the question whether the languages of nondeterministic LBAs are closed under complement was open for three decades before being answered affirmatively by Immerman and Szelepcsényi independently.

We need a more stringent condition, something more restrictive than just linear space.

Space Constraints



A primitive decision algorithm. It is convenient to separate the input (read-only) of size n from the (read-write) worktape, which has size $s(n)$.

Little Space

One might suspect that we get less and less compute power as we decrease the memory-size function $s(n)$, say, to $\log n$, $\log\log n$, $\log\log\log n$, and so on.

Here is a major surprise:

Theorem (Hartmanis, Lewis, Stearns 1965)

Suppose some decision problem is not solvable in constant space. Then every Turing machine solving the problem requires space $\Omega(\log\log n)$ *infinitely often.*

Hence, once we get to $s(n) = o(\log\log n)$ we might as well have a worktape of fixed size. The proof is somewhat complicated, see the website.

From Fixed to None

But allowing only a constant amount of work tape is already equivalent to allowing no work tape at all: there are only finitely many possible work tape contents and head positions.

These finitely many worktape configurations can be coded as part of the finite state control.

In other words, we move the fixed size tape into the control unit of the Turing machine and we get Zero Space.

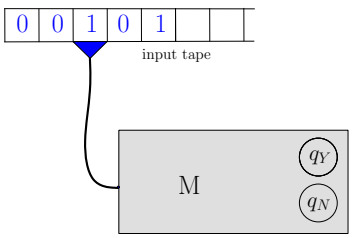
Zero Space

Zero Space sounds more impressive, but since we can store a limited amount of information in the state of the machine, it might be better to talk about Constant Space.

Again, we do not charge for the size of the input on the read-only tape; the strings there can be arbitrarily long.

Also, for the time being we will focus on decision algorithms, so we don't need an output tape: we can use special Yes/No states to indicate acceptance. This is the old distinction between acceptors versus transducers, between decision problems versus function problems.

No Worktape



Left-To-Right

From the definition of a Turing machine, the read-only input tape can be scanned repeatedly and the tape head may move back and forth over it.

As it turns out, one can assume without loss of generality that the read head only moves from left to right only: at each step one symbol is scanned and then the head moves right and never returns.

Theorem (Rabin/Scott, Shepherdson)

Every decision problem solved by a constant space two-way machine can already be solved by a constant space one-way machine.

The proof of this result is quite messy, and we won't go into details. See the website.

Simplified Configurations

Note that configurations for these restricted Turing machines are simpler than in the general case, all we need is

$$p\,x \qquad p \in Q, x \in \Sigma^\star$$

There is no need to keep track of the "left" part of the tape, we can never go back there.

One step in the computation is then given by a map δ, the so-called transition function, where

$$p\,ax \;\vdash_M^1\; q\,x \iff \delta(p,a) = q$$

Acceptance

The computation on input x ends after exactly $|x|$ steps in some state p without any input left.

There is no need for a special halting state, we can simply read off the "response" of the machine by inspecting the last state.

If this state is "good" we think of the machine as having accepted its input; otherwise it rejects.

Example: Parity and Majority

Let's suppose the input is given as a bit sequence $x = x_1 x_2 \ldots x_{n-1} x_n$. Here are two classical problems concerning these sequences:

- **Parity:** Is the number of 1-bits in x even?
- **Majority:** Are there more 1-bits than 0-bits in x?

Parity requires just one bit: just add the bits in x modulo 2.

On the other hand, Majority seems to require an integer counter of unbounded size $\log n$ bits; we will see in a while that Majority indeed cannot be solved in zero space.

Parity Checker


```
s = 0;

while( there is another input bit x )
       s = x xor s;

return s;
```

This really computes the exclusive-or of all the bits, which happens to be the right answer:

$$s = x_1 + x_2 + \ldots + x_{n-1} + x_n \pmod 2$$

Streaming Algorithms

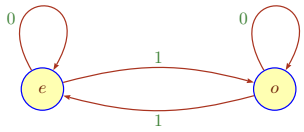
We can think of these restricted Turing machines as algorithms that read each bit in an input stream just once, perform a very simple operation after each bit is read, and return the answer after the last bit was processed.

```
initialize;

while( there is another input bit x )
       process x;      // state transition

return answer;
```

The algorithm updates its internal state after scanning a new bit (performs a state transition).

Transition Diagrams

An excellent representation for our parity checker is a diagram:



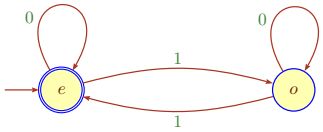
The edges are labeled by the input bits, and the nodes indicate the internal state of the checker (called e and o for clarity, these are the states of the Turing machine).

Note that similar diagrams don't work well for ordinary Turing machines.

This pictures are very easy to read and interpret for humans (and useless for computers).

Complete Information

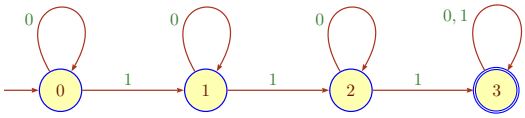
It is customary to indicate the initial state by a sourceless arrow, and the so-called final states states (corresponding to answer Yes) by marking the nodes.



In this case state e is both initial and final.

"Final state" is another example of bad terminology, something like "accepting state" would be better. Alas ...

Another Example

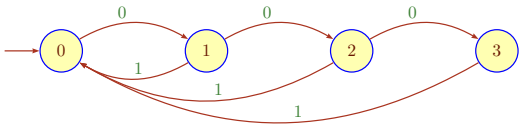


There are 4 internal states $\{0, 1, 2, 3\}$ and input x will take us from state 0 to state 3 if, and only if, it contains at least 3 1-bits.

Initial state is 0 and 4 is the sole final state.

Run-length Limited Codes

Consider all binary words with the property that all 1-bits are separated by between 1 and 3 0-bits.



Here all states are considered accepting.

Note that there is no transition labeled 0 out of state 3: this is an incomplete automaton, it "crashes" on input 0000.

Checking Small Divisors

A typical primality testing algorithm starts very modestly by making sure that the given candidate number x is not divisible by small primes, say, 2, 3, 5, 7, and 11.

Assume n has 1000 bits. Using standard arithmetic to do the tests is not particularly smart, we want a very fast method to eliminate lots of bad candidates quickly.

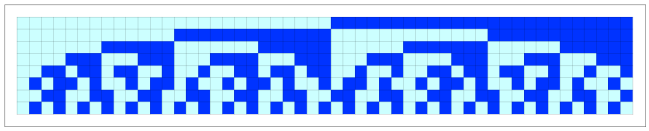
One could customize the division algorithm for a small divisor d but even that's still clumsy.

Can we use a scan algorithm?

Mod 5 Base 2

Numbers up to 250 in binary that are divisible by 5 (written here in columns, MSD on top).

Note the regularity of the bit patterns.



Dire Warning: At first glance, it looks like there is self-similarity in this picture. There isn't, at least not in any technical sense.

Pictures can be dangerous.

Induction to the Rescue

Write $\nu(x)$ for the numerical value of bit-sequence x, assuming the MSD is read first.

Then

$$\nu(x0) = 2 \cdot \nu(x)$$
$$\nu(x1) = 2 \cdot \nu(x) + 1$$

So if we are interested in divisibility by, say, $d = 5$ we have

$$\nu(xa) = 2 \cdot \nu(x) + a \pmod 5$$

Since we only need to keep track of remainders modulo 5 there are only 5 values, corresponding to 5 internal states of the loop body.

Table Lookup

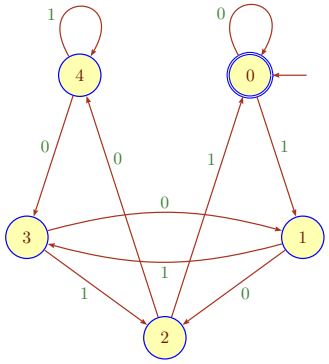
In most implementations, the operation ν would be precomputed into a lookup table.

In the actual run all that's needed is then a simple table lookup, depending on the current state, and the next bit.

	0	1	2	3	4
0	0	2	4	1	3
1	1	3	0	2	4

The precomputation may be costly in general (it's not in this particular case), but once we have the table performance will be excellent.

Remainders Mod 5



Diagrams

The last picture is really a (particularly pretty) layout of a directed graph associated with the machine:

- The nodes of the graph are the states of the machine.
- The edges correspond to transitions and are labeled by letters.

Note that strictly speaking we have to allow multiple labels: different input symbols may cause the same transition from one state to another. It is coincidence that this situation does not arise in the last example.

Alternatively, we could allow for multiple edges and label each with exactly one symbol. Conceptually, there is little difference between the two approaches but note that implementation details could vary quite a bit.

On occasion one also ignores the labels and just deals with the underlying digraph.

Optimality in Time

Lower bound arguments are often tricky, but this really is the fastest possible algorithm for divisibility by 5 as can be seen by an adversary argument.

Suppose there is an algorithm that takes less than n steps.

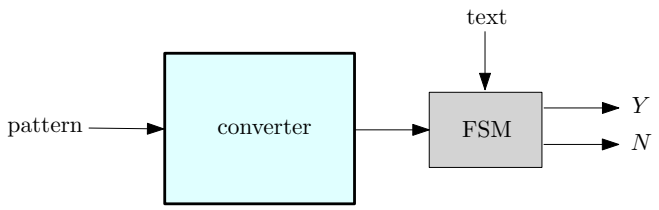
Then this algorithm cannot look at all the bits in the input, so it will not notice a single bit change at least one particular place.

But that cannot possibly work, every single bit change in a binary number affects divisibility by 5:

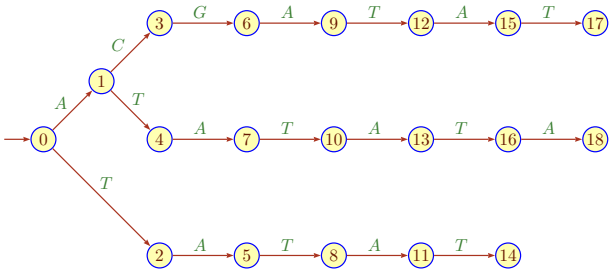
$$x \pm 2^k \neq x \pmod 5$$

for any $k \geq 0$.

The Killer App

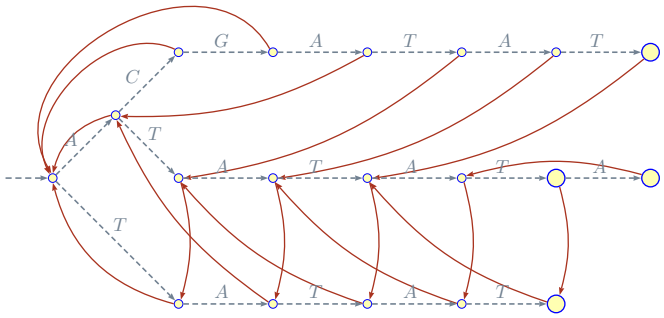


Text Search



This machine searches for strings ACGATAT, ATATATA and TATAT.

Back-Transitions



The Algorithms

We will shortly discuss closure properties of the languages associated with finite state machines. It will follow from these general results that a machine searching for words like ACGATAT, ATATATA and TATAT trivially exists.

The important point is that there are algorithms that construct the machines very efficiently, given the words as input. For example, the algorithm used in the last example is due to Aho and Corasick.

These algorithms can be quite sophisticated and clever; there is a whole field referred to as stringology that deals with them.

However, we will focus on another important algorithmic application of finite state machines.

Two Threads

There are two somewhat separate reasons as to why finite state machines are hugely important.

- They can be used for text processing purposes, and are lightning fast in this capacity. There is no word processor nor any compiler that does not use FSMs.
- The can be used in a variety of decision algorithms. These algorithms tend to be more complicated and less efficient but are still very important for example in model checking.

The Machine Perspective

We can think of our devices as consisting of two parts:

- a transition system, and
- an acceptance condition.

The transition system includes the states and the alphabet and can be construed as a labeled digraph.

Definition

A transition system or semi-automaton (SA) is a structure

$$\langle Q, \Sigma, \delta \rangle$$

where Q and Σ are finite sets and $\delta \subseteq Q \times \Sigma \times Q$.

The elements of δ are transitions and often written in suggestively as $p \xrightarrow{a} q$.

Sequences, Words, Strings

It is customary to refer to the input sequences as words or strings.

Given an alphabet Σ one writes Σ^* for the collection of all words over Σ, and Σ^+ for the collection of all non-empty words.

In practice, the alphabet is usually one of

- $\mathbf{2} = \{0, 1\}$ (2)
- $\{0, 1, \ldots, 9\}$ (10)
- $\{0, 1, \ldots, 9, A, \ldots, F\}$ (16)
- lowercase letters (26)
- ASCII (128 or 256)
- UTF-8 (1,112,064)

but it is better to keep the definition general. Very large alphabets cause interesting algorithmic problems.

Runs

Suppose $\mathcal{A}$ is some semi-automaton. Given a word $u = a_1 a_2 \ldots a_m$ over the alphabet of $\mathcal{A}$ a run of the automaton on u is an alternating sequence of states and letters

$$p_0, a_1, p_1, a_2, p_2, \ldots, p_{m-1}, a_m, p_m$$

such that $p_{i-1} \xrightarrow{a_i} p_i$ is a valid transition for all i. p_0 is the source of the run and p_m its target, and $m \geq 0$ its length. So a run is just a path in a labeled digraph.

Sometimes we will abuse notation and also refer to the corresponding sequence of states alone as a run:

$$p_0, p_1, \ldots, p_{m-1}, p_m$$

Traces

Given a run

$$\pi = p_0, a_1, p_1, a_2, p_2, \ldots, p_{m-1}, a_m, p_m$$

of an automaton, the corresponding sequence of labels

$$a_1 a_2 \ldots a_{m-1} a_m \in \Sigma^\star$$

is referred to as the trace or label of the run.

Every run has exactly one associated trace, but the same trace may have several runs, even if we fix the source and target states (ambiguous automata).

Nondeterminism

So, a transition system is just an edge-labeled digraph where the labels are chosen from some alphabet.

In the spirit of Rabin/Scott's 1959 paper, it is perfectly acceptable to have nondeterministic transitions

$$p \xrightarrow{a} q \quad \text{and} \quad p \xrightarrow{a} q'$$

Note that these transitions are somewhat problematic from a "real algorithm" perspective: are we supposed to go to q or to q'?

This idea may sound quaint today, but it was a huge conceptual breakthrough at the time.

Special Semi-Automata

Definition

A semi-automaton is complete if for all $p \in Q$ and $a \in \Sigma$ there is some $q \in Q$ such that

$$p \xrightarrow{a} q$$

is a transition.

In other words, the system cannot get stuck in any state.

Definition

A semi-automaton is deterministic if for all $p, q, q' \in Q$ and $a \in \Sigma$

$$p \xrightarrow{a} q, p \xrightarrow{a} q' \quad \text{implies} \quad q = q'$$

Thus, a deterministic system can have at most one run from a given state for any input.

Acceptance Conditions

The acceptance condition depends much on the automaton in question but it is always a condition on the runs associated with a word u.

The (acceptance) language $\mathcal{L}(\mathcal{A})$ of the automaton $\mathcal{A}$ is the set of all words accepted by the automaton.

The most basic kind of acceptance condition is comprised of an initial state q_0 and a collection of final states $F \subseteq Q$.

A run is accepting if it starts at q_0 and ends in some state in F.

This corresponds to the idea of resetting the automaton to state q_0 before the computation starts, ignoring all intermediate steps, and using only the last state to determine acceptance.

DFAs

Combining the previous acceptance condition with completeness and determinism produces a particularly useful type of automaton.

Definition

A deterministic finite automaton (DFA) is a structure

$$\mathcal{A} = \langle Q, \Sigma, \delta; q_0, F \rangle$$

where $\langle Q, \Sigma, \delta \rangle$ is a deterministic and complete semi-automaton and $q_0 \in Q$, $F \subseteq Q$.

It is straightforward to see that a DFA has exactly one trace (or run) on any possible input word.

We use the standard acceptance condition: a run is accepting if it leads from q_0 to some $q \in F$.

Extending the Transition Function

It is often convenient to think of the transition function as a map $\delta : Q \times \Sigma^\star \to Q$ defined by primitive recursion over words:

$$\delta(p, \varepsilon) = p$$
$$\delta(p, xa) = \delta(\delta(p, x), a)$$

In terms of the extended transition function acceptance can be expressed easily:

$\mathcal{A}$ accepts a word u iff $\delta(q_0, u) \in F$.

Note that for all words x and y:

$$\delta(p, xy) = \delta(\delta(p, x), y)$$

Regular Languages

Definition

A language $L \subseteq \Sigma^\star$ is recognizable or regular if there is a DFA M that accepts L: $\mathcal{L}(M) = L$.

Thus a regular language has a simple, finite description in terms of a particular type of finite state machine. As we will see, one can manipulate the languages in many ways by manipulating the corresponding machines.

In a sense, regular languages are the simplest kind of languages that are of interest (there are more complicated types of languages such as context-free languages that are critical for computer science).

No Majority

The diagram perspective is useful to show that the Majority language $M = \{ x \in \mathbf{2}^\star \mid \#_0 x < \#_1 x \}$ is not regular.

For assume otherwise and let n be the number of states in a DFA accepting M. By definition, $0^n 1^{n+1}$ is accepted.

But then there is a path from q_0 to a final state q, labeled $0^n 1^{n+1}$.

The first part must contain a loop that we can traverse multiple times, leading to an accepted input of the form $0^m 1^{n+1}$ where $m > n + 1$.

Contradiction.

This rather trivial observation is also known at the Pumping Lemma (for regular languages).

Again: Killer Apps

1. Membership in a regular language can be tested blindingly fast, and using only sequential access to the letters of the word. This works very well with streams and is the foundation of many text searching and editing tools (such as `grep` and `emacs`). All compilers use similar tools.
2. Another important aspect is the close connection between finite state machines and logic. Here we don't care so much about acceptance of particular words but about the whole language. The truth of a formula can then be expressed as "some machine has non-empty acceptance language." Actually, this becomes really interesting for infinite words (where the first application disappears entirely).

First, a bit of basic theory.

Fast Acceptance Testing

Proposition

For any DFA M and any input string x we can test in time linear in $|x|$ whether M accepts x, with very small constants.

```
p = q0;                    // reset
while(  a = x.next() )     // next input symbol
      p = delta[p][a];     // table look-up

return  p in F;            // table look-up
```

Of course, it might take some time to compute the lookup table δ in the first place, but once we have it acceptance testing is very fast.

Example: Divisibility Testing DFA

Example

$$\mathcal{A} = \langle \{0, \ldots, 4\}, \{0, 1\}, \delta; 0, \{0\} \rangle$$

where the transition relation, written as a function $\Sigma \times Q \to Q$, is

$$\delta = \begin{pmatrix} 0 & 2 & 4 & 1 & 3 \\ 1 & 3 & 0 & 2 & 4 \end{pmatrix}$$

As we have seen, this DFA checks whether a binary number has numerical value divisible by 5:

$$\mathcal{L}(\mathcal{A}) = \{ x \in \mathbf{2}^\star \mid \nu(x) = 0 \pmod 5 \}.$$

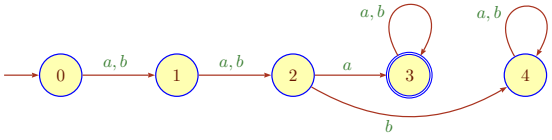
Note that the terminology is actually quite bad: a DFA should really be called a "deterministic complete finite automaton" or DCFA.

Should, but isn't. Once bad terminology is widely accepted there is no way to get rid of it.

Example: Checking a Letter

A machine that accepts all words over alphabet $\{a, b\}$ that have an a in the third position.

$$\mathcal{A} = \langle \{0, \ldots, 4\}, \{a, b\}, \delta; 0, \{3\} \rangle$$



Note that state number 4 is superfluous in a way.

Sinks and Traps

Definition

A state p in a DFA is a trap if for all symbols a: $\delta(p, a) = p$.

A state in a DFA is a sink if it is a trap and is not final.

Note that we could remove a sink from a DFA without changing the acceptance language. However, this would break the completeness condition (though not determinism).

This is really an implementation detail; on occasion completeness is important and other times it is not.

Warning: some authors allow incomplete machines under the name DFA to accommodate sink removal. We will always refer to these devices as partial DFAs (PDFAs) or incomplete DFAs.

The Membership Problem

Given any language one is faced with a natural decision problem: determine whether some word belongs to the language. In this particular case the language is represented by a DFA.

Problem:	**DFA Membership (DFA Recognition)**
Instance:	A DFA M and a word x.
Question:	Does M accept input x?

Lemma

The DFA Membership Problem is solvable in linear time.

As we will see, there are other representations for regular languages where the membership problem is more difficult to solve. This is of great practical importance; many pattern matching problems can be phrased as membership in regular languages but using descriptions that are more difficult to deal with than DFAs.

More Decision Problems

Apart from membership testing there are several more complicated decision problems associated with finite state machines that have efficient solutions as long as the machine is a DFA. Again, these are crucial in many applications.

Problem:	**Emptiness**
Instance:	A DFA M.
Question:	Does M accept any input string?

Problem:	**Finiteness**
Instance:	A DFA M.
Question:	Does M accept only finitely many strings?

Problem:	**Universality**
Instance:	A DFA M.
Question:	Does M accept all input strings?

Easy Decidability

Theorem

The Emptiness, Finiteness and Universality problem for DFAs are decidable in linear time.

Proof.

Consider the unlabeled diagram G of the machine. Emptiness means that there is no path in G from q_0 to any state in F, a property that can be tested by standard linear time graph algorithms (such as DFS or BFS). □

Exercise

Show how to deal with Finiteness and Universality.

Optimality in Size

A general problem related to computation that we have not yet encountered is program size complexity:

What is the (size of the) smallest program for a given task?

Note that this is somewhat orthogonal to the usual time and space complexity of an algorithm: here the issue is the size of the code, not it's efficiency. Can you program a SAT checker on your wrist watch?

In general identifying smallest programs is very hard. In particular for Turing/register machines the problem is highly undecidable.

But for DFAs there is a very good solution.

Equivalence and State Complexity

It is easy to see that the same language can be recognized by many different machines.

Definition

Two DFAs M_1 and M_2 over the same alphabet are equivalent if they accept the same language: $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

Given a few equivalent machines, we are naturally interested in the smallest one. In some sense, the smallest machine is the best representation of the corresponding regular language.

Definition

The state complexity of a DFA is the number of its states.

The state complexity of a regular language L is the size of a smallest DFA accepting L.

Existence

Note that the state complexity of a regular language always exists, albeit for a silly reason: the natural numbers are well-ordered.

However, there are two potential problems that could make a smallest machine somewhat useless.

- There might be several DFAs of minimal size.
- Even if there is only one (up to isomorphism), larger DFAs for the same language might have no reasonable connection to the minimal one.

The first problem would make it difficult to compare languages on the basis of their smallest machines.

The second problem could make it difficult to obtain a smallest machine given an arbitrary one.

We will see that for DFAs neither problem occurs.

Computing State Complexity

Naturally there is a computational problem lurking in the dark:

Problem:	**State Complexity**
Instance:	A regular language L.
Solution:	The state complexity of L.

As we will see, state complexity is always computable but for efficient solutions we need L to be represented by a DFA.

Note that we could ask similar questions for Turing machines (Kolmogorov-Chaitin complexity). Alas, in this general setting everything becomes highly undecidable. For example, one cannot determine the smallest Turing machine that writes a given target string on the tape and then halts.

State Complexity: An Example

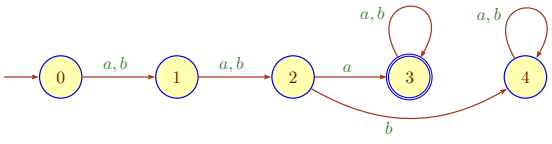
There are good algorithms to calculate the state complexity of a given regular language (unless it is so large that we cannot actually build a DFA for it), so state complexity becomes really interesting only when we consider a class of languages.

For example, one might ask what is the state complexity of the languages

$$L_{a,k} = \{\, x \in \{a,b\}^\star \mid x_k = a \,\}.$$

Thus $x \in L_{a,k}$ iff the kth symbol in x is an a.

For positive k this is not a problem: we can just skip over the first $k-1$ symbols and then verify that x_k really is a.



The Nasty Case

But what if k is negative?

Meaning that we are looking for the $|k|$th symbol from the end. E.g,

$$L_{a,-3} = \{aaa, aab, aba, abb, aaaa, aaab, aaba, aabb, \ldots\}$$

The crucial problem here is that the DFA does not know ahead of time when the last input will appear. We can't just go backwards from the end.

This may seem like a preposterous restriction, but streams do behave just like this; we don't know when the last input will come along.

Exercise

Figure out the state complexity of $L_{a,k}$ for negative k. No strict lower bound is required at this point, just come up with a machine that feels best possible.

Numbers and DFAs

Here is a much harder problem that deals with standard radix representations of integers.

Write $\nu_B(x)$ or simply $\nu(x)$ for the numerical value of string x written in base B, so $x \in \{0, 1, \ldots, B-1\}^\star$.

Also, one has to be a bit careful about the MSD and LSD. Unless otherwise noted, we assume that the MSD is the first digit, so

$$\nu(x_k x_{k-1} \ldots x_1 x_0) = \sum_{i \le k} x_i \, B^i.$$

If the LSD is first we have a reverse radix representation.

We already know that divisibility by a fixed number m can be tested by a DFA with respect to base $B = 2$. But there are many other, useful numeration systems and is is not entirely clear whether one can build DFAs for all of them.

Divisibility in Base B

Lemma

Divisibility by m can be tested by a DFA in any base B.

Proof.

We can construct a canonical Horner automaton for this task.

Keep the state set $Q = \{0, 1, \ldots, m-1\}$.

Change the transition function to

$$\delta(p, a) = p \cdot B + a \pmod m.$$

Initial state and only final state is 0.

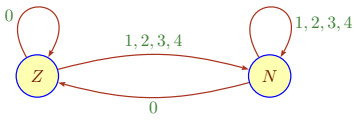
Since $\delta(q_0, x) = \nu(x) \pmod m$ this works.

□

How About State Complexity?

But note that that in some special cases this construction is not very clever.

For example, to check whether a number written in base 5 is divisible by 5 the canonical solution looks like this:



This makes it tempting to consider the state complexity of divisibility languages: what is the smallest possible DFA that recognizes one of these languages?

Experimental Data

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	2	3	3	2	3	3	2	3	3	2	3	3	2	3
4	3	4	2	4	3	4	2	4	3	4	2	4	3	4	2
5	5	5	5	2	5	5	5	5	2	5	5	5	5	2	5
6	4	3	4	6	2	6	4	3	4	6	2	6	4	3	4
7	7	7	7	7	7	2	7	7	7	7	7	7	2	7	7
8	4	8	3	8	5	8	2	8	5	8	3	8	5	8	2
9	9	3	9	9	4	9	9	2	9	9	4	9	9	4	9
10	6	10	6	3	6	10	6	10	2	10	6	10	6	3	6
11	11	11	11	11	11	11	11	11	11	2	11	11	11	11	11
12	5	5	4	12	3	12	4	5	7	12	2	12	7	5	4
13	13	13	13	13	13	13	13	13	13	13	13	2	13	13	13
14	8	14	8	14	8	3	8	14	8	14	8	14	2	14	8
15	15	6	15	4	6	15	15	6	4	15	6	15	15	2	15
16	5	16	3	16	8	16	3	16	9	16	5	16	9	16	2

$m :\downarrow$, $B :\rightarrow$.

Data Mining

The problem is to extract useful information from this table.

Unfortunately, there aren't too many patterns that are clearly visible.

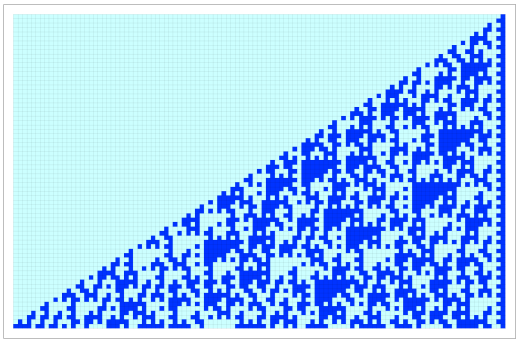
- For m a prime things seem straightforward.
- Base $B = 2$ seems potentially doable (but not obvious).

The problem was solved a few years ago by a high-school student (Boris Alexeev, 2nd place Intel STS 2004). Alas, the result is not very pretty.

More later when we have the right tools available.

How about more complicated properties of numbers?

Suppose we want to recognize powers of 3 written base 2.



This looks rather complicated. In fact there is no DFA that could recognize these numbers (but the proof is quite hard, see Cobham's theorem).