

CDM

Memoryless Machines

Klaus Sutner

Carnegie Mellon University

Fall 2011

Outline

- 1 Constant Space
- 2 Finite State Machines
- 3 DFA Decision Problems

Structuring Computation

We have seen that self-similarity can be used as an organizing principle to structure computation via recursion (top-down) or iteration (bottom-up).

Here is another interesting principle: memoryless computation.

This name is a white lie, what we really mean is computation with bounded memory, but that does not sound anywhere near as sexy. For example, Floyd's cycle finding algorithm is memoryless in this sense.

Very Concrete Computability

It is now customary to identify feasible computation with polynomial time computation. There are some obvious problems with this identification but overall it has turned out to be a very productive idea.

Question: Is there a reasonable class of algorithms that are

- interesting and useful in the **RealWorld**TM, and
- as primitive as they can possibly be?

So we are looking for some kind of “atomic” algorithms.

Full Disclosure

One could argue that the most basic yet interesting class of algorithms can be interpreted as *Boolean functions* (or circuits if you like):

$$f : 2^n \rightarrow 2^m$$

For example, fixed precision integer multiplication falls into this class of problems.

Implementing these algorithms, even for fairly modest values of n and $m = 1$, turns out to be very difficult and leads to many interesting data structures (such as binary decision diagrams), as well as nice theoretical results (classifications of Boolean functions via clones).

True, but let's focus on algorithms with infinitely many instances.

Scaling Down

One natural starting point would be to impose restrictions on register machines. For example, we could insist that registers only can hold k -bit integers (and modify the increment operation accordingly).

All fine and good, but it is slightly more convenient to switch to **Turing machines** at this point: the process of reading the input is non-trivial on a Turing machine and we will attach the actual computation to it.

Exercise

Figure out how to modify register machines to obtain a useful notion of a k -bit register machine.

Knowledge Transfer

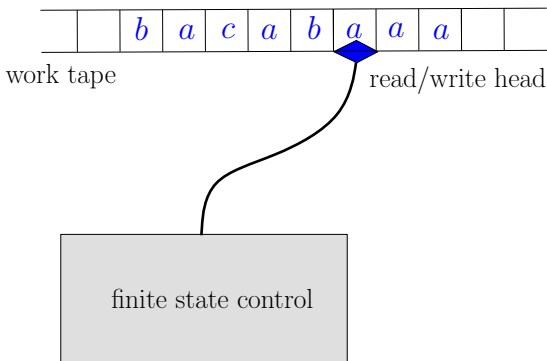
The following quote is attributed to S. Banach:

“Good mathematicians see analogies between theorems or theories; the very best ones see analogies between analogies.”

We won't go through the details of defining complexity for register machines, all the important ideas carry over from Turing machines (YAMC: yet another model of computation).

Total Recall: Turing Machines

A **Turing machine** is a finite state control that has access to (potentially bi-infinite tape) via a read/write head.



With a tiny bit of effort one can show that register machines and Turing machines are computationally equivalent in the sense that a function is RM-computable iff it is TM-computable.

Bounding Turing Machines

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

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

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 restriction leads to an important class of machines: **linear bounded automata (LBA)** and the corresponding class $\text{SPACE}(n)$ of problems solvable in linear space.

Note that the running time of an LBA is necessarily at most exponential: otherwise a configuration would have to repeat. A function computable on an LBA is primitive recursive at a very low level.

In fact, because the running time bound is easily computable one can see that

Proposition

It is decidable whether a LBA accepts its input x .

Emptiness

Alas, LBA are still way too powerful to serve as “atomic” algorithms.

It is not hard to see that an LBA can check whether a given input codes an accepting computation of a Turing machine. Write a sequence of configurations such as

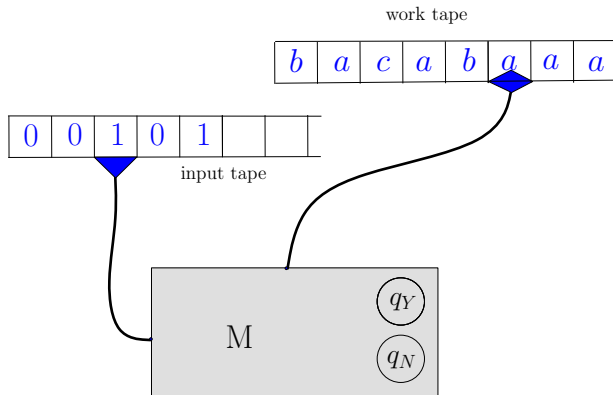
$$\# C_0 \# C_1 \# C_2 \dots \# C_{n-1} \# C_n \#$$

on the tape of the LBA. The LBA can then easily check that C_i really leads to C_{i+1} in one step, and that C_0 and C_n are appropriate.

But then the Emptiness Problem (see below) is undecidable for LBAs: otherwise we could solve the Halting Problem. No good for our purposes.

A Radical Proposal

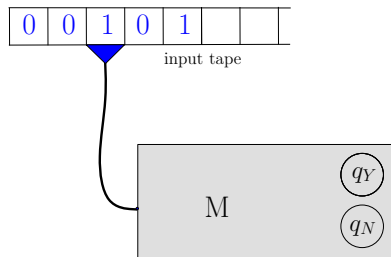
Here is a really drastic resource constraint: $\text{SPACE}(1)$



Constant is Zero

Note that we can think of the contents of a fixed size worktape and the head position as part of the state of the machine.

Hence we don't need the worktape at all.



Getting Rid of Zig-Zags

The input tape is read-only, but we are allowed to make multiple passes over the input.

As it turns out, one pass is enough.

Theorem (Rabin, Scott; Shepherdson)

We may assume without loss of generality that the input head moves left-to-right only.

Of course, the one-way machine may have a larger state set than the original two-way one.

The proof is quite messy (uses so-called crossing sequences).

Rabin and Scott

In 1959, Rabin and Scott wrote the seminal paper in automata theory

M. Rabin, D. Scott
Finite Automata and Their Decision Problems
IBM Journal of Research and Development
Volume 3, Number 2, Page 114 (1959)

This paper introduces nondeterminism and the systematic study of decision problems associated with finite state machines.

Broken Turing Machines



If all transitions are of the form

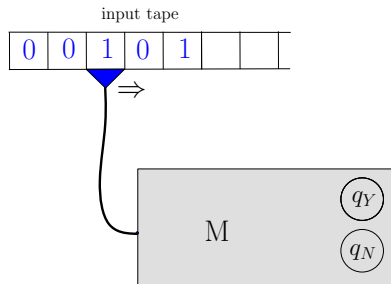
$$(p, a) \mapsto (q, a, +1)$$

then the machine simply reads the input in a single left-to-right scan, changing its state as it goes along.

It never changes the tape inscription, and when it reaches the end of the input we can only look at the state to distinguish between an accepting and non-accepting computation.

So Where Are We?

We have a read-only, left-to-right input tape and a finite state control:



For the time being, we only consider acceptors, later we will also consider transducers of this type.

Simplified Configurations

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

$$p x \quad p \in Q, x \in \Sigma^*$$

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

$$p a x \mid_M q x$$

whenever state p scanning symbol a leads to state q .

Transition Function

This leads to a simplified lookup table for the machine, all we need is a function

$$\delta : Q \times \Sigma \rightarrow Q$$

Of course, we also need an initial state.

And we need to modify our acceptance mechanism slightly: For general Turing machine acceptors one usually postulates two special halting states q_Y and q_N indicating acceptance and rejection, respectively.

In our scan-once setting it is convenient to allow for multiple accepting states, the so-called **final states**. Also, we won't insist that they are halting: if there is another input symbol the computation simply continues.

The Algorithm Perspective

Here is another way of looking at these machines. The input is a sequence of letters in some fixed alphabet:

$$x = x_1x_2x_3 \dots x_{n-1}x_n$$

The length n here is variable and we do not assume to know what it is ahead of time.

Think of an input stream: we can keep extracting the next input bit until, for the first time, the extraction operation fails.

Scan-Once Algorithms

We consider algorithms that read each input letter just once, perform a very simple operation after each letter is read, and return the answer after the last letter was processed.

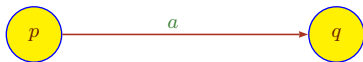
```
initialize;  
  
while( there is another input letter x )  
    process x;    // perform state transition  
  
return answer;
```

The algorithm updates its internal state after scanning the next input symbol.

Transition Diagrams

As before with register machines, a good representation for these machines is a diagram (for general Turing machines these diagrams are a bit messy, but here they work well).

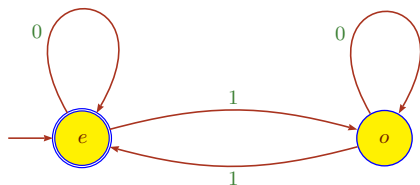
We represent a transition $\delta(p, a) = q$ by a labeled edge



It is customary to indicate the initial state by a sourceless arrow, and final states by marking the nodes.

Example: Parity Checker

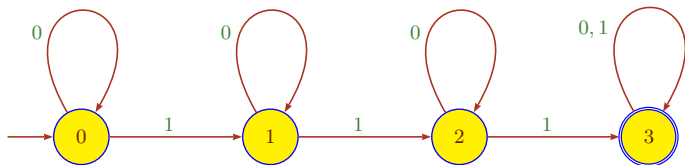
Here is a little 2-state machine that accepts its binary input iff it contains an even number of 1's.



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

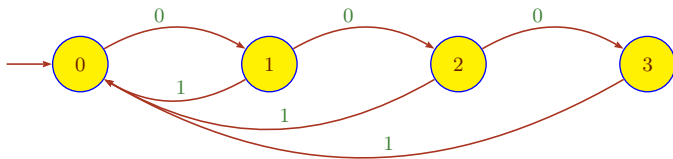


There are 4 internal states $\{0, 1, 2, 3\}$ and input x will take us from state 0 to state 3 if, and only if, it contains at least 3 1-bits.

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 (incomplete automaton).

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.

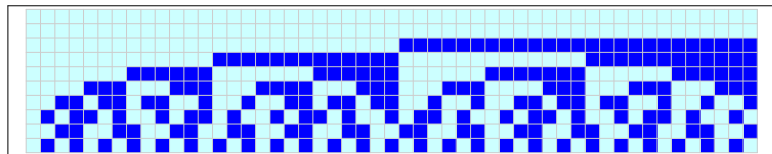
One could customize the division algorithm for, say, divisor 5 but even that's still clumsy.

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.

Beware of hasty conclusions.

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 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 states of the loop body.

Table Lookup

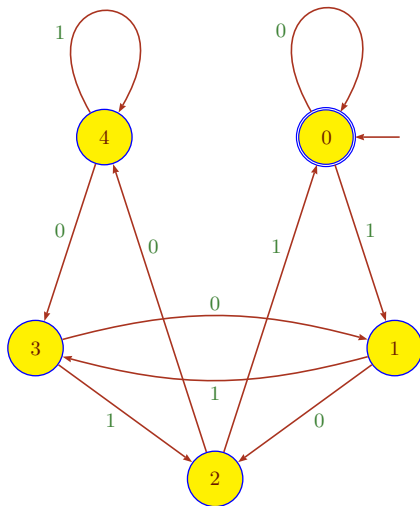
In most implementations, the operation ν would be pre-computed into a lookup table.

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 pre-computation 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



Optimality in Time

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 alleged algorithm cannot look at all the bits in the input, so it will not notice a single bit change in the right place.

But that cannot possibly work since a single bit change in a binary number changes divisibility:

$$x \pm 2^k \not\equiv x \pmod{5}$$

for any $k \geq 0$.

- Constant Space

2 Finite State Machines

- DFA Decision Problems

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

- $\mathbf{2} = \{0, 1\}$
- $\{0, 1, \dots, 7\}$
- $\{0, 1, \dots, 9\}$
- $\{0, 1, \dots, 9, A, \dots, F\}$
- lowercase letters
- ASCII (128 or 256)

but it is better to keep the definition general. Some constructions are based on very large alphabets.

Traces and Runs

Suppose \mathcal{A} is some semi-automaton. Given a word $u = a_1a_2 \dots a_m$ over the alphabet of \mathcal{A} a **trace** of the automaton on u is an alternating sequence of states and letters

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

such that $p_i \xrightarrow{a_i} p_{i+1}$ is a valid transition for all i . The integer $m \geq 0$ is the length of the trace.

The corresponding sequence of states alone

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

is a **run** of \mathcal{A} on u

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 $a \in \Sigma$

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

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 .

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.

DFA's

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 $q \in F$. Using the extended transition function from above this means

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

Example: Divisibility Testing DFA

Example

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

where the transition relation, written as a function $\Sigma \times Q \rightarrow Q$, 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:

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

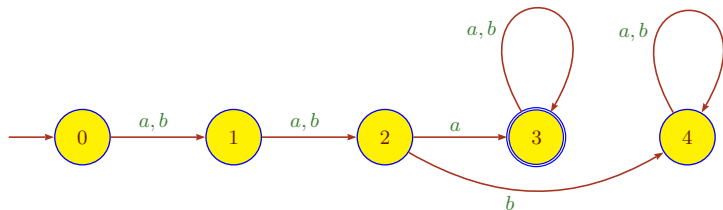
Note that the terminology is actually quite bad: a DFA should really be called a “deterministic complete finite automaton” or DCFA.

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.

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



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** or **incomplete DFAs**.

The Iteration Perspective

We can think of the transition function δ of a DFA as a list of Σ -many functions from states to states. For $a \in \Sigma$ consider

$$\begin{aligned}\delta_a &: Q \rightarrow Q \\ \delta_a(p) &= \delta(p, a)\end{aligned}$$

We can “iterate” these functions according to some input word

$u = u_1u_2 \dots u_n$:

$$\delta_u = \delta_{u_n} \circ \delta_{u_{n-1}} \circ \dots \circ \delta_{u_2} \circ \delta_{u_1}$$

It follows that

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

The Algebra Perspective

Alternatively we can think of these functions as generating a sub-monoid T of $Q \rightarrow Q$, the so-called **transformation monoid**, where the operation is functional composition, as always. One way of writing down T is

$$T = \{ \delta_x : Q \rightarrow Q \mid x \in \Sigma^* \}$$

where $\delta_x(p) = \delta(p, x)$.

This approach may seem overly abstract and convoluted but it turns out to be quite useful. FSMs are rather unique in this sense: for more complicated types of machines algebra is not as directly useful.

More Algebra: Semigroups and Monoids

Definition

A **semigroup** is a set together with an associative operation. If there is also a neutral element we speak of a **monoid**.

For words (or, if you prefer, sequences), the natural operation is **concatenation**. If we exclude the empty word we have a semigroup under concatenation, and a monoid if we include it.

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

Again, we are just talking about sequences of atomic events, first a , then b , then a , ...

Semigroup Actions

Thinking of Σ^* as a semigroup it is tempting to construe the transition function as a kind of “multiplication”. The lookup table defines the multiplication on single letters:

$$Q \times \mathbf{2} \longrightarrow Q$$

This multiplication can naturally be extended to the whole monoid by iterating the table lookup like so:

$$Q \times \mathbf{2}^* \rightarrow Q$$

$$p \cdot \varepsilon = p$$

$$p \cdot xa = (p \cdot x) \cdot a$$

This type of operation is called a **(right monoid) action**. Later we will see left actions and actions over groups; they turn out to be hugely important for certain counting problems.

Regular Languages

Definition

A language $L \subseteq \Sigma^*$ 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).

The Killer Apps

Why should one care about DFAs and regular languages?

- One important reason is that 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`).
- 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." More about this later.

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.

- Constant Space
- Finite State Machines
- ③ DFA Decision Problems

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 infinitely 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). \square

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 its efficiency. Can you program a SAT checker on your wrist watch?

In general identifying smallest programs is very hard. In particular for Turing 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.

State Complexity: An Example

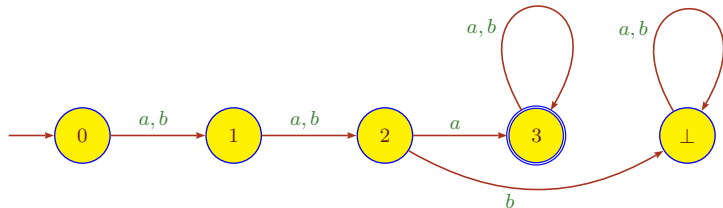
There are good algorithms to calculate the state complexity of a given regular language (unless it is so large that we cannot actually build a DFA for it), so state complexity becomes really interesting only when we consider a class of languages.

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

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

Thus $x \in L_{a,k}$ iff the k th 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,

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

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 $x \in \{0, 1, \dots, B - 1\}^*$.

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

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

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 it 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 $Q = \{0, 1, \dots, m - 1\}$.

Change the transition function to

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

Initial state and only final state is 0.

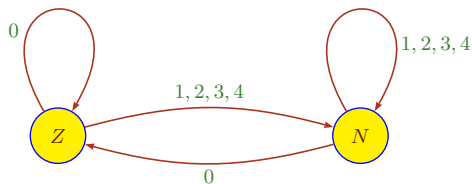
Since $\delta(q_0, x) = \nu(x) \pmod{m}$ this works.



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

 $m : \downarrow \quad B : \rightarrow$

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).

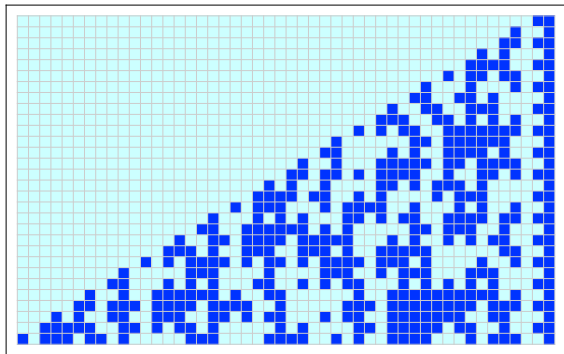
The problem was solved a few years ago by a high-school student (Boris Alexeev, 2nd place Intel STS 2004). Alas, the result is not very pretty.

More later when we have the right tools available.

Hard Question

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 hard).

Summary

- DFAs are yet another model of computation, a very limited type of linear time, constant space computation.
- One can think of a DFA as a “best possible” algorithm for the recognition of certain classes of strings.
- Some properties of DFAs such as Emptiness, Finiteness and Universality are easily decidable in linear time.
- The number of states of a DFA is a measure of its complexity (an example of program size complexity).
- Divisibility by a (or several) fixed modulus can be checked by a DFA regardless of base.
- It can be quite difficult to determine the state complexity of a family of regular languages.