CDM

Memoryless Machines

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Weak Turing Machines
- Fast Scan Algorithms
- DFAs and Regular Languages
- State Complexity
- Divisibility Testing
- Product Machines
- Accessibility

Scan Algorithms

Very Concrete Computability

So far we have seen a description of abstract computability in terms of Turing 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 a better match.

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.

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

$$\# x_1 x_2 \ldots x_{n-1} x_n \#$$

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

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.

More Constraints

Unfortunately, LBA are still much too powerful to admit a good structure theory. To get really simple devices we need much stronger restrictions.

Here is a really drastic idea:

- Allow the tape head only to move to the right.

Since we cannot go back to read any symbol again, this also means in essence that our machine is read-only.

Since the head movement is so constrained there is also no need for end markers: we simply scan one symbol x_i after the other, changing state as we go along.

When the last symbol is read the "response" of the machine is determined by the current state.

Exercise 1. *Explain why read-only is not a restrictions if we already do not allow the head to move left.*

The Algorithm Perspective

Here is another motivation:

Question:
What are the fastest possible algorithms?
Does this notion make any sense?

We may safely assume that the input is a sequence of bits (or, more generally, letters in some fixed alphabet)

$$x = x_1 x_2 x_3 \ldots x_{n-1} x_n$$

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.

Some algorithms work in constant time: "check if the first bit is 1".

This type of problem is clearly not very interesting.

Scanning All Bits

How about questions involving more or less all the bits in the input?

Let's insist that the algorithm scans each bit and returns a true/false answer when the last bit is reached.

Classical examples

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

Majority seems to require an integer counter of unbounded size $\log n$ bits.

But Parity really requires just one bit.

Parity Checker

$O(1)$ space is enough for Parity: just count modulo 2.

```
s = 0;

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

return s;
```

Really computes the exclusive-or of all the bits (which happens to be the answer):

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

Scan Algorithms

We are looking for algorithms that read each input bit 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;
```

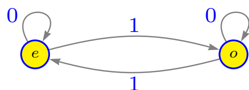
The algorithm updates its internal state after scanning a new bit (*state transition*).

Crucial constraint:

The whole computation must be constant space, (and hence each step constant time).

Transition Diagrams

An excellent representation for our parity checker is a diagram:



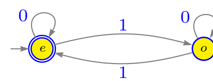
The edges are labeled by the input bits, and the nodes indicate the internal state of the checker (called e and o for clarity, these are the states of the Turing machine).

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

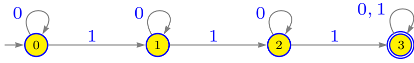
It is customary to indicate the initial state by a sourceless arrow, and the so-called *final states* states (corresponding to answer Yes) by marking the nodes.



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

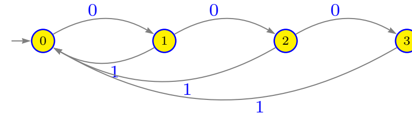


There are 4 internal states $\{0, 1, 2, 3\}$ and input x will take us from state 0 to state 3 if, and only if, it contains at least 3 1-bits.

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 (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.

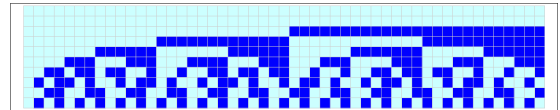
One could customize the division algorithm for divisor 5 but even that's still clumsy.

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.

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 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

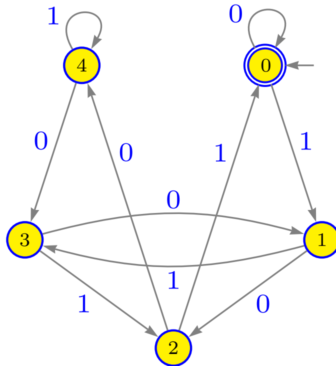
In most implementations, the operation ν would be pre-computed into a lookup table.

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



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

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.

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

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

for any $k \geq 0$.

Finite State Machines

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 1. *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

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

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

Traces and Runs

Suppose $\mathcal{A}$ is some semi-automaton. Given a word $u = a_1 a_2 \dots a_m$ 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 *(acceptances) 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 said to be 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 2. *A deterministic finite automaton (DFA) is a structure*

$$M = \langle Q, \Sigma, \delta; q_0, F \rangle$$

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

Note that this is really an amputated Turing machine: since the head moves right by one cell at each step, there is no need to specify head movement or new tape symbols.

In a DFA the transition relation is really a function $\delta : Q \times \Sigma \to Q$.

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

Example

Example 1.

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

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

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

Thus, this DFA checks whether a binary has numerical value a multiple of 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.

Sinks and Traps

Definition 3.

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 require a DFA to be both complete and deterministic.

Extending the Transition Function

Since traces in a DFA are unique the last state in a run is uniquely determined by the initial state and the scanned word. We can express this by iterating the transition function:

$$\delta : Q \times \Sigma^\star \to Q$$
$$\delta(p, \varepsilon) = p$$
$$\delta(p, xa) = \delta(\delta(p, x), a)$$

where x is a word and a a single letter.

We can then express the acceptance condition as follows:

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

Fast Acceptance Testing

Proposition 1. *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;            // tabel 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.

Regular Languages

Definition 4. *A language $L \subseteq \Sigma^\star$ is recognizable or regular if there is a DFA M that accepts L: $\mathcal{L}(M) = L$.*

Note that we are using a slightly strange approach here: usually one first defines a class of functions (Turing computable, primitive recursive, polynomial time computable, . . .) and then the corresponding relations via characteristic functions.

This time we have no functions, only languages (representing the Yes-instances of some decision problem).

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**
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 1. *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 2. *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 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 5. *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 6. *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

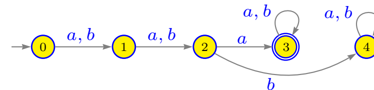
There are good algorithms to calculate the state complexity of a given regular language (unless it is so large that we cannot actually build a DFA for it), so state complexity becomes interesting only when we consider a class of languages.

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 3. *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

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

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 1. *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.

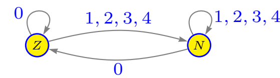
Since $\delta(q_0, x) = \nu(x) \pmod m$ this works.

□

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:



Question: Can we compute the state complexity of the divisibility languages?

What is the size of the smallest DFA that checks divisibility by m when the input is given in radix B (or in reverse radix B)?

Experimental Data

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

Data Mining

The problem is to extract useful information from this table.

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

- Base $B = 2$ seems doable.
- For m a prime things also seem straightforward.

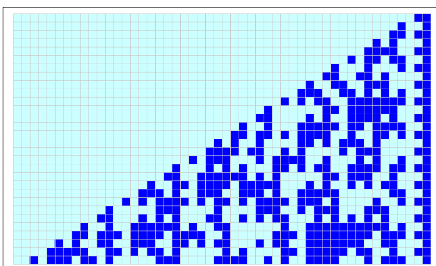
The problem was recently solved by a high-school student (Boris Alexeev, 2nd place Intel STS 2004).

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.

Constructing Automata

Multiple Moduli

Since we can't quite handle the original question, how about this one:

Can we check divisibility by two moduli m_1 and m_2?

Of course, we could run two DFAs sequentially on the same input. But the corresponding algorithm would be outside of the realm of FSMs. Can we do this in just one run?

Sure, we could check for divisibility by $\mathrm{lcm}(m_1, m_2)$.

Generalize: we actually want a machine for $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.

Question:
Is there a more systematic way to do this?

Products of DFA

Suppose we have two DFAs M_1 and M_2 over the same alphabet Σ.

Construct a new DFA $M = M_1 \times M_2$, a so-called *product machine*, as follows.

$$Q = Q_1 \times Q_2$$
$$\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$$

The new initial state is (q_{01}, q_{02}).

So the new machine M essentially runs M_1 and M_2 in parallel.

Acceptance Conditions for a Product Machine

What inputs are accepted by the product machine?

Depends on our choice of final states.

We can get union and intersection of $\mathcal{L}(M_1)$ and $\mathcal{L}(M_2)$ by setting

union $\quad F = F_1 \times Q_2 \cup Q_1 \times F_2$

intersection $\quad F = F_1 \times F_2$

Exercise 4. *Explain why these two choices work.*

Closure Properties

Note that we have established a closure property for regular languages:

Lemma 2. *Regular languages are closed under union and intersection.*

More importantly, we have effective closure: give two DFAs we can easily compute a new DFA for $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ and $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$.

This should sound very familiar by now: there is an analogous result for (Turing) decidable relations and for primitive recursive relations.

How About Complement?

Can we build a DFA for $\Sigma^\star - \mathcal{L}(M)$?

This is actually even easier: we modify the acceptance condition by replacing F by its complement:

$$F' = Q - F$$

in the given DFA.

Theorem 2. *The regular languages are closed under Boolean operations union, intersection and complement.*

The closure is effective: there is an algorithm to construct the corresponding finite state machines.

Deciding Equivalence

There is another decision problem lurking in the dark:

Problem: **Equivalence**
Instance: Two DFAs M_1 and M_2.
Question: Are the two machines equivalent?

We can solve this problem now by a product machine construction:

Lemma 3. *M_1 and M_2 are equivalent iff $\mathcal{L}(M_1) - \mathcal{L}(M_2) = \emptyset$ and $\mathcal{L}(M_2) - \mathcal{L}(M_1) = \emptyset$.*

Note that the lemma yields a quadratic time algorithm. We will see a better method later.

Deciding Inclusion

Observe that we actually are solving two instances of a closely related problem here:

Problem: **Inclusion**
Instance: Two DFAs M_1 and M_2.
Question: Is $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$?

which problem can be handled by

Lemma 4. $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$ *iff* $\mathcal{L}(M_1) - \mathcal{L}(M_2) = \emptyset$.

Note that for any class of languages Equivalence is decidable when Inclusion is so decidable. However, the converse may be false – but it's not so easy to come up with an example.

Recognizing Words

Given any word $w \in \Sigma^\star$ it is trivial to construct a DFA M_w such that

$$\mathcal{L}(M_w) = \{w\}\,.$$

It follows from our closure properties that we can build a DFA for any finite set of words:

$$\mathcal{L}(M) = \{w_1, w_2, \ldots, w_s\}\,.$$

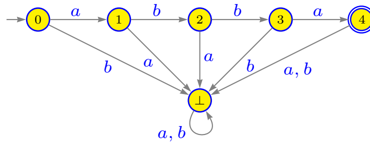
But how about efficiency: the machines for single words have essentially the same size as the word, so they are optimal in a sense.

On the other hand, the machine for a set of words might be quite large; we have to be a bit careful about how to construct the machine.

State Complexity

The natural DFA M_w has $|w| + 2$ states.

For example, let $w = abba$.



State 0 is initial, and 4 is final. $\perp$ is a sink state.

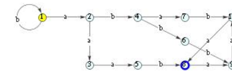
Exercise 5. *Show that $|w| + 2$ is indeed the state complexity of $\{w\}$.*

Complicated Intersections

The last example is a bit of an abuse of the product construction; there are better ways to build DFAs for finite languages.

But in general, product constructions are indispensible: it can be quite difficult to build automata for, say, the intersection of two regular languages directly by hand.

Here is an example: build a DFA for the language of all words that contain the subword (not factor) ab 3 times, and a multiple-of-3 number of a's. Building the two component machines and taking their product we get



Sizes of Product Machines

Suppose machine M_i has n_i states.

Then the product machine

$$M = M_1 \times M_2 \times \ldots M_{s-1} \times M_s$$

has $n = n_1 n_2 \ldots n_s$ states.

That's a disaster! The product machine grows exponentially. More precisely, if the state complexity of $\mathcal{L}(M)$ really is n this is fine. But otherwise we would like a construction that produces a smaller machine.

Just think about a DFA that recognizes all the reserved words in a standard programming language.

We need methods to keep the size of our machines at or close to the state complexity of the corresponding language.

Accessible Part

Definition 7. *A state p in a DFA is* accessible *if $\delta(q_0, x) = p$ for some word x. The automaton is accessible if all its states are.*

Thus a state is accessible it can be reached from the initial state by a sequence of transitions.

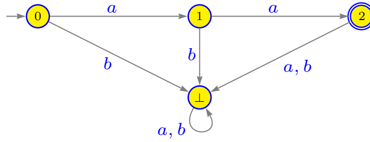
Now suppose we remove all the inaccessible states from a DFA M.

After adjusting Q, δ and F we obtain a new DFA M', the so-called *accessible part* of M.

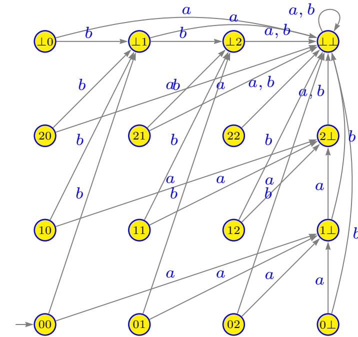
Lemma 5. *The machines M and M' are equivalent.*

Example

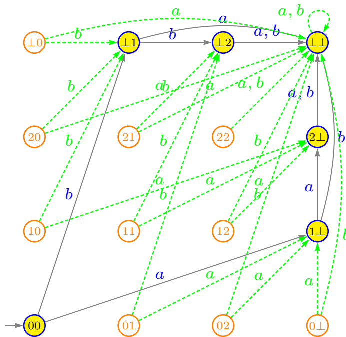
Consider the product automaton for M_{aa} and M_{bb}.



Full Product Automaton



The Accessible Part



Constructing the Accessible Part

So suppose we are given a DFA M and we want to construct its accessible part M'.

Testing whether a state is accessible is really a path existence problem in the diagram of the machine. Routinely solved in linear time by standard graph exploration algorithms (DFS, BFS).

Lemma 6. *The accessible part of a DFA can be constructed in linear time.*

This is important for machines constructed by other algorithms, not those built by hand – no one would ever write down inaccessible states (hopefully).

Avoiding Inaccessibility

Much more interesting is the following problem: how do we make sure that algorithms (such as the product machine construction) do not produce any inaccessible states to begin with?

This is yet another instance of a closure problem: we have to find the least subset of the full product automaton that contains (q_{01}, q_{02}) and is closed under out-going transitions.

This is very similar to the problem of, say, generating a subgroup of a given group.

We will postpone our discussion of closure operations a bit.

Summary

- DFAs are yet another model of computation, a very limited type of linear time 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).
- The associated regular languages are closed under union, intersection and complement.
- This closure is effective: we can construct the corresponding machines in quadratic time and space.
- 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.
- Accessibility is closely related to generating a structure.