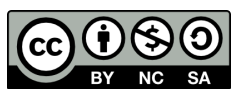
CDM

Memoryless Machines

Klaus Sutner

Carnegie Mellon University

Spring 2024



1 **Zero Space**

2 **Finite State Machines**

3 **The Foundations**

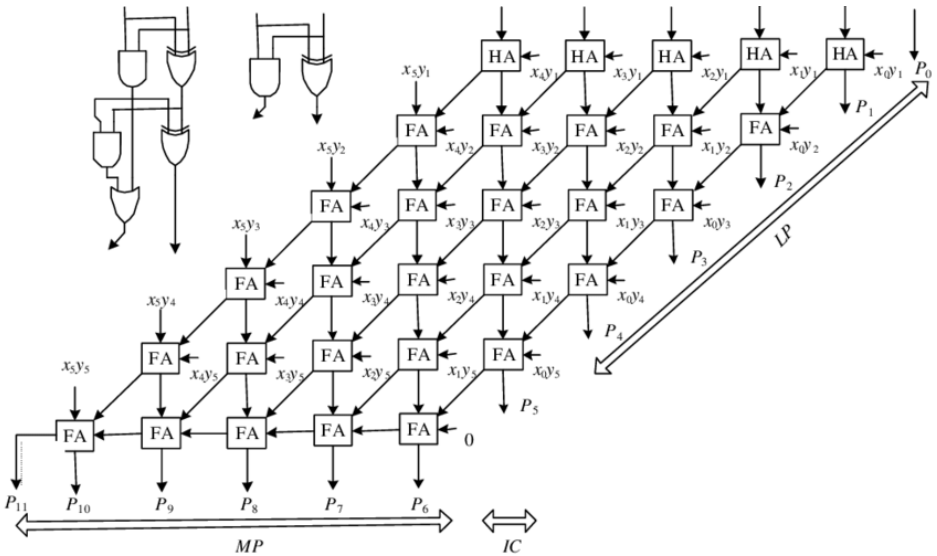
So far, we have blissfully ignored physical limitations. For example, we pretend that our registers can hold arbitrarily large naturals, and increment/decrement them in one step.

We could avoid this by limiting the registers to k-bit numbers for some fixed k, say, $k = 64$. So we wind up with a particularly bad assembly language.

In our model, this has an unintended side-effect: there are only finitely many possible inputs (recall: the input has to be written into registers).

This is in stark contrast with standard arithmetical problems like primality testing.

Algorithms that take as input some fixed, finite number of bits and return a similar output are hugely important.



Alas, this leads to a different realm (Boolean circuits, gates, electrical engineering) that we do not want to get involved with.

Claim: Every finite decision problem is decidable in constant time (albeit for entirely the wrong reasons). Similarly every finite function can be computed in constant time.

More precisely, we can simply hardwire a lookup table that lists the correct answer for each instance.

The lookup table may get so large that this brute-force approach is not practical (think about multiplying two 64-bit numbers), but from the perspective of recursion theory everything is trivial. We need an input domain like $\mathbb{N}$ or $\mathbf{2}^\star$ for our machinery to kick in.

The last claim is perfectly correct, but it really means that we need to develop a better framework. Clearly, we can distinguish different levels of difficulty even for finite problems. Think about a finite set of instances I, say, all 1000-bit numbers. Determine

- all numbers in I divisible by 17,
- all prime numbers in I,
- all $x \in I$ such that $\{x\}() \downarrow$.

Everyone would agree that these problem are listed in order of increasing complexity, the first one is nearly trivial, the second requires a bit of work and the last one is a hot mess.

Convention:
We will only consider problems with infinite instance sets.

It is also a good idea to give up on the domain of arithmetic and only talk about natural numbers. We could use sequence numbers to code any conceivable finitary structure, but that requires overhead that obscures the details of simple computations.

Experience shows that a switch to strings or words over some finite alphabet works well. So the standard input domain will be $\Sigma^\star$ and in particular $\mathbf{2}^\star$.

Note that Turing machines naturally work on strings, so we are drifting towards a different model of computation.

Recall: PR String Functions

Recall from HW that one can define a clone of primitive recursive string functions that has essentially the same power as ordinary p.r. arithmetic functions.

At this point, we want to dive down to string functions that are very, very easy to compute.

How could we define an easily computable class of functions that still has interesting properties and applications?

To process a string we clearly have to read it, say, from left to right. How about limiting the algorithm to just that?

Of course, if we read the string we could simply store it into memory and the perform an arbitrarily complicated computation. To prevent that, we take a radical step:

> We can only use a constant amount of memory.

For this to make sense we do not charge the algorithm for the memory needed to store the input: the algorithm sees a stream of letters from some finite alphabet Σ (think in particular of the binary alphabet $\mathbf{2}$).

Let us refer to the memory contents as the state of the algorithm, so there are only finitely many states.

When the input string has been consumed, the algorithm makes a decision based solely on the last state (we will see how to adjust this model to computing a function later).

If this sort of decision algorithm return Yes, we say that it accepts or recognizes the input string.

The collection $L \subseteq \Sigma^\star$ of all such strings is the acceptance language of or the language recognized by the algorithm.

These languages are called regular or (finite state) recognizable.

A sexier name for this kind of restricted computation is zero space.

This terminology is weakly justified by the fact that one can often nicely distinguish between the internal states of a system and its memory (think about register machines).

Our algorithms have only state but no memory at all, so we have the class of zero-space problems.

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 can easily be handled without memory: 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.

```
s = 0;          // initialize state

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

return s;
```

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

$$s = x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \oplus x_n$$

So this is an extremely simple case of a streaming algorithm, when the number of scans is just 1 and the memory is constant (as opposed to a small number of scans, using little memory).

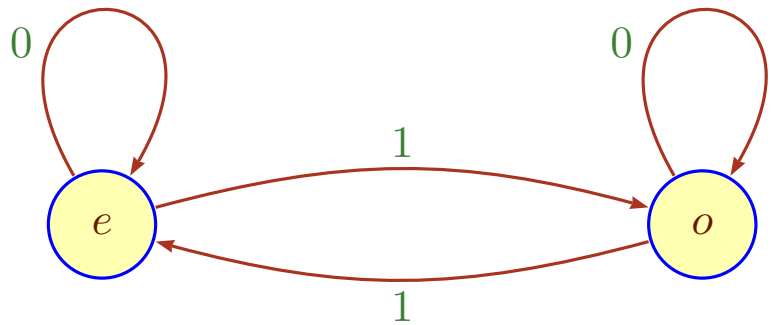
```
initialize;

while( there is another input letter x )
       process x;          // update system

return answer;
```

The point is that the state transition is extremely fast, typically using a lookup table, or evaluating a simple function.

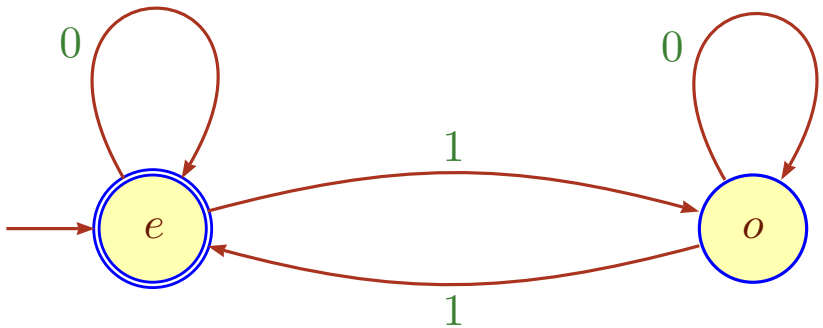
A most useful representation for our parity checker is a diagram:



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 two internal states).

This pictures are very easy to read and interpret for humans (and useless as input to algorithms).

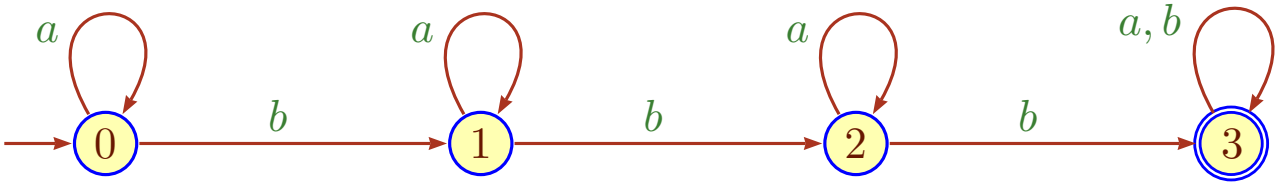
It is customary to indicate the initial state (where all computations start) by a sourceless arrow, and the so-called final states states (corresponding to answer Yes) by marking the nodes.



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 . . .

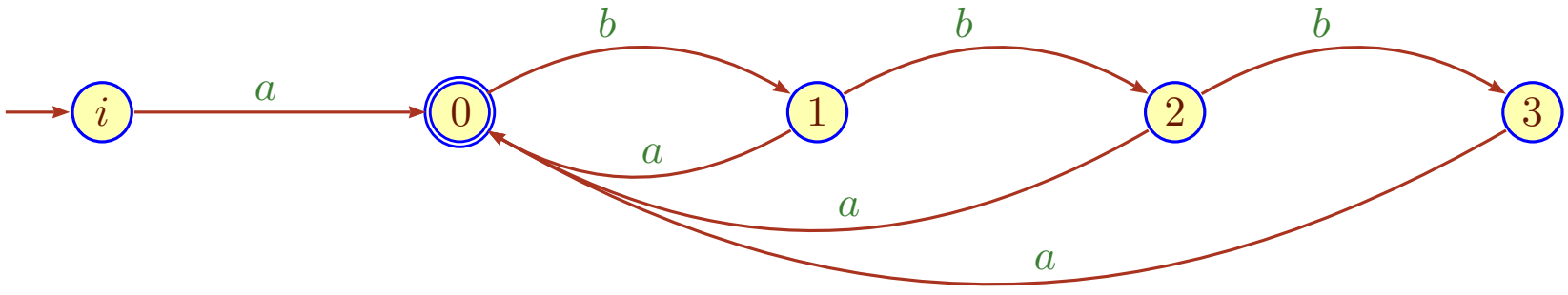
Another Example



There are 4 states $\{0, 1, 2, 3\}$. Input $x \in \{a, b\}^\star$ will take us from state 0 to state 3 if, and only if, it contains at least 3 letters b.

The "correctness proof" here consists of staring at the picture for a moment.

Consider all words over $\{a, b\}$ that start and end with a and have the property that all as are separated by 1, 2 or 3 bs.



We allow missing transitions: if the machine reads b in state i it simply "crashes" (see the formal definition of acceptance below). As a practical matter, partial transition functions are critical for efficiency.

Correctness is by diagram chasing. Note that the informal description above does not explain whether the first and last a need to be distinct. Deal with the other case.

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 (actually, checking the first 100 or so primes seems to be more realistic in practice).

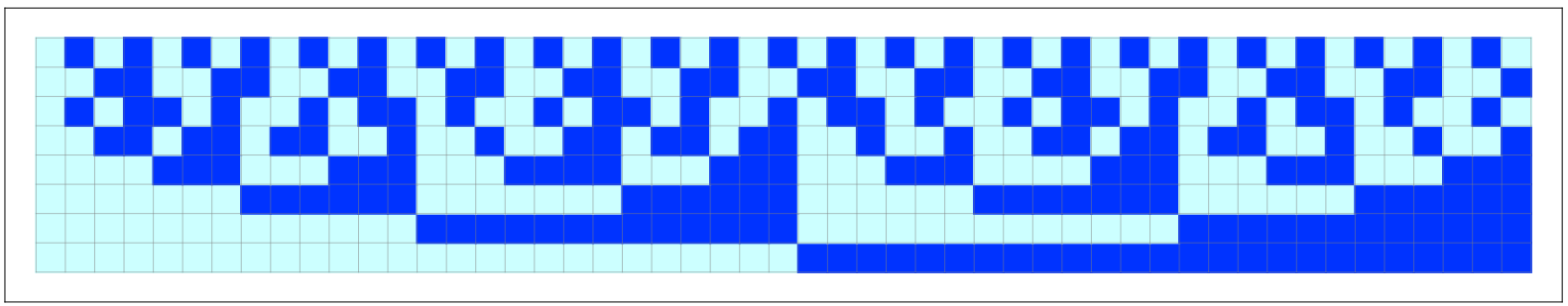
Assume n has 1000 bits. Using standard large integer library to do the tests is not really a good idea, we want a very fast method to eliminate lots of bad candidates quickly.

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

Can we use one of our memoryless machines?

8-bit binary numbers that are divisible by 5 (written here in columns, LSD on top).

There is some regularity in the bit patterns, but it's elusive.



We need a machine that accepts these bit pattern, but rejects all others. And, of course, works for an arbitrary number of bits.

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

Then

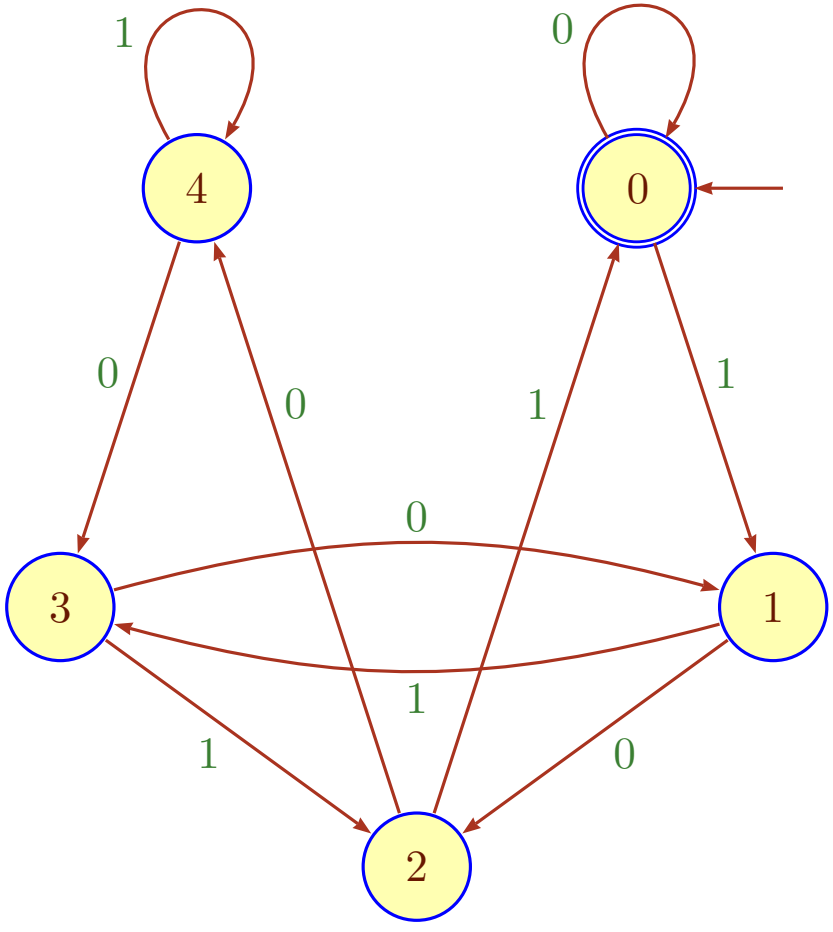
$$\nu(x0) = 2 \cdot \nu(x)$$

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

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.


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

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 in 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 $k \geq 0$.

And Majority?

Majority cannot be handled by a our finite state algorithms.

For suppose otherwise. Suppose q_0 is the initial state and start feeding the algorithm 0s. By the old lasso argument, we must encounter some state twice:

$$q_0 \xrightarrow{0^a} p \xrightarrow{0^b} p$$

Hence, the algorithm must accept input $0^{a+b}1^{a+b+1}$. But then it also accepts $0^{a+2b}1^{a+b+1}$, contradiction.

In fact, we could repeat the loop $p \to p$ any number of times, producing infinitely many inputs where the algorithm fails.

1 Zero Space

2 **Finite State Machines**

3 The Foundations

We can think of our string decision algorithms as a sort of machine 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 that we will refer to as the diagram of the automaton.

Definition

A transition system is a structure

$$\langle Q, \Sigma, \tau \rangle$$

where Q and Σ are non-empty finite sets (the state set and the alphabet) and $\tau \subseteq Q \times \Sigma \times Q$ is the transition relation of the structure. The elements of τ are transitions and often written $p \xrightarrow{a} q$.

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

In practice, the alphabet is usually along the lines of

- digit alphabets: binary $\mathbf{2} = \{0, 1\}$, decimal, hexadecimal
- letter alphabets: ASCII (subset thereof),
- large alphabets: UTF-8, $\mathbf{2}^k$ product alphabet

Algorithmically, there is a major difference between small and large alphabets, a difference we will mostly ignore.

Fix some transition system $\mathcal{A} = \langle Q, \Sigma, \tau \rangle$. Given a word $u = a_1 a_2 \dots a_m$ over Σ, a run of $\mathcal{A}$ on u is an alternating sequence of states and letters

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

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:

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

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).

Definition

A finite state machine (FSM) or finite automaton (FA) is a structure

$$\mathcal{A} = \langle \mathcal{T}; \mathrm{acc} \rangle$$

where $\mathcal{T} = \langle Q, \Sigma, \tau \rangle$ is a transition system and acc is an acceptance condition.

We will make no attempt to define the concept of an acceptance condition in general and simply explain various examples as we go along.

The acceptance condition determines whether an FA accepts or recognizes some input $x \in \Sigma^\star$.

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

The most basic acceptance condition is comprised of a collection of initial states $I \subseteq Q$ and a collection of final or accepting state $F \subseteq Q$.

Vanilla acceptance:
A run is accepting if it starts in I and ends in F.

Note that this condition does not require knowledge of the whole run, we just need to worry about the first and last state.

We can also think of finite state machines as a basic model of computation. Recall that in order to define how such a model works, one defines configurations, snapshots that contains all the information needed to resume the computation later. In this case, we only need to keep track of the current state $p \in Q$ and the remainder $z \in \Sigma^\star$ of the input.

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

One step in a computation is then given by τ, really just a lookup table.

$$p\,az \;\vdash_{\mathcal{A}}^{1}\; q\,z \iff \tau(p, a, q)$$

Here $p \in Q$, $a \in \Sigma$, $z \in \Sigma^\star$.

On this view, the computation on input x ends after exactly $|x|$ steps in some state q without any input left. We accept if that state is final:

$$p\,x \vdash_{\mathcal{A}} q \qquad p \in I, q \in F$$

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

Decidability[†] here comes down to being able to recognize a particular language $L \subseteq \Sigma^\star$: we want a FSM $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$.

[†]We don't have any computable functions here, just decision algorithms. And there is no issue with termination.

Useful States

Only those states in a finite state machine are relevant that lie on a path from I to F. A state is called accessible if it is reachable from I, and coaccessible if F is reachable from it.

A state p is a trap if all transitions with source p also have target p.

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

One uses similar terminology for the whole automaton. In particular, an automaton is trim if it is both accessible and coaccessible.

One also speaks of the accessible/coaccessible/trim part of a FSM.

More precisely, one can use standard graph exploration algorithms to compute the accessible part of an automaton: determine the states reachable from I in the diagram, and remove all the others (and the transitions affected by this). It is clear that this can be done in linear time.

Coaccessible and trim parts are computed similarly.

Claim: Let $\mathcal{A}$ be a FSM and $\mathcal{A}'$ its trim part. Then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

Definition

A transition system is complete if for all $p \in Q$ and $a \in \Sigma$ there is some $q \in Q$ and a transition

$$p \xrightarrow{a} q$$

In other words, the system cannot get stuck in any state, we always can consume all input symbols and obtain a run of length $|x|$.

Definition

A transition system 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.

A deterministic transition system consists of a collection of partial functions

$$\delta_a : Q \rightharpoonup Q$$

where $a \in \Sigma$. By currying, we can also think of these as a map

$$\delta : Q \times \Sigma \rightharpoonup Q$$

We will refer to these maps as transition functions.

If the transition system is in addition complete, we get plain functions $\delta_a : Q \to Q$ and $\delta : Q \times \Sigma \to Q$.

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

Definition

A partial deterministic finite automaton (PDFA) is a structure

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

where the transition system $\langle Q, \Sigma, \delta \rangle$ is deterministic. If the system is in addition complete we call the structure a deterministic finite automaton (DFA). We use the vanilla acceptance condition (path from q_0 to F).

It is straightforward to see that a PDFA has at most one trace (or run) starting at q_0, of length at most $|x|$, on any possible input word x.

For a DFA there is exactly one trace starting at q_0, of length $|x|$, on any possible input word x.

Arguably, DFAs should be called *complete, deterministic finite automata*, acronym CDFA. Unfortunately, no one does this.

We will refer to nondeterministic machines of all kinds as NFAs.

Note that one can safely assume that all states in a DFA are accessible: we can replace the automaton by its accessible part (in linear time).

This fails for coaccessibility: the coaccessible/trim part of a DFA may just be PDFA. DFAs are nicer in many ways, but from an algorithmic perspective PDFAs rule the roost.

Definition

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

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

In a sense, recognizable languages are the simplest kind of languages that are of interest. More complicated types of languages such as context-free/context-sensitive languages are critical for compilers and complexity theory, but even recognizable languages are surprisingly powerful.

*Regular is more popular in the US, but hopelessly overloaded.

Note that we are using a slightly strange approach here: usually one first defines a class of functions (RM computable, primitive recursive, polynomial time computable, . . .).

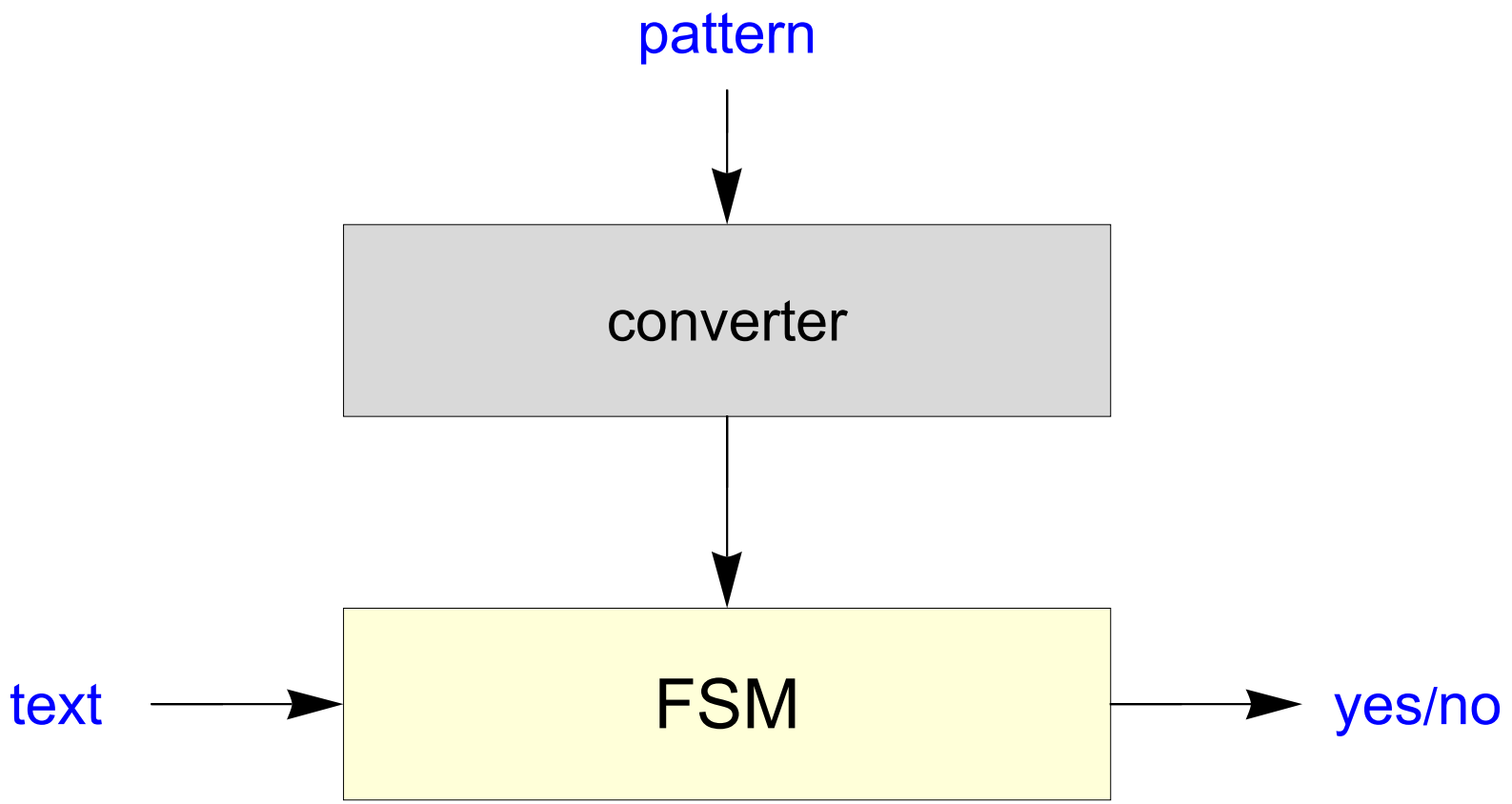
Then one introduces the corresponding class of decision problems via characteristic functions. This time we have no functions, only languages.

There is a class of finite state machines that compute functions, so-called transducers, that require a bit more effort to deal with. More later.

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

1. Membership in a recognizable 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).


pattern
converter
text
FSM
yes/no

Proposition

For any DFA $\mathcal{A}$ and any input string x we can test in time linear in $|x|$ whether $\mathcal{A}$ 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
```

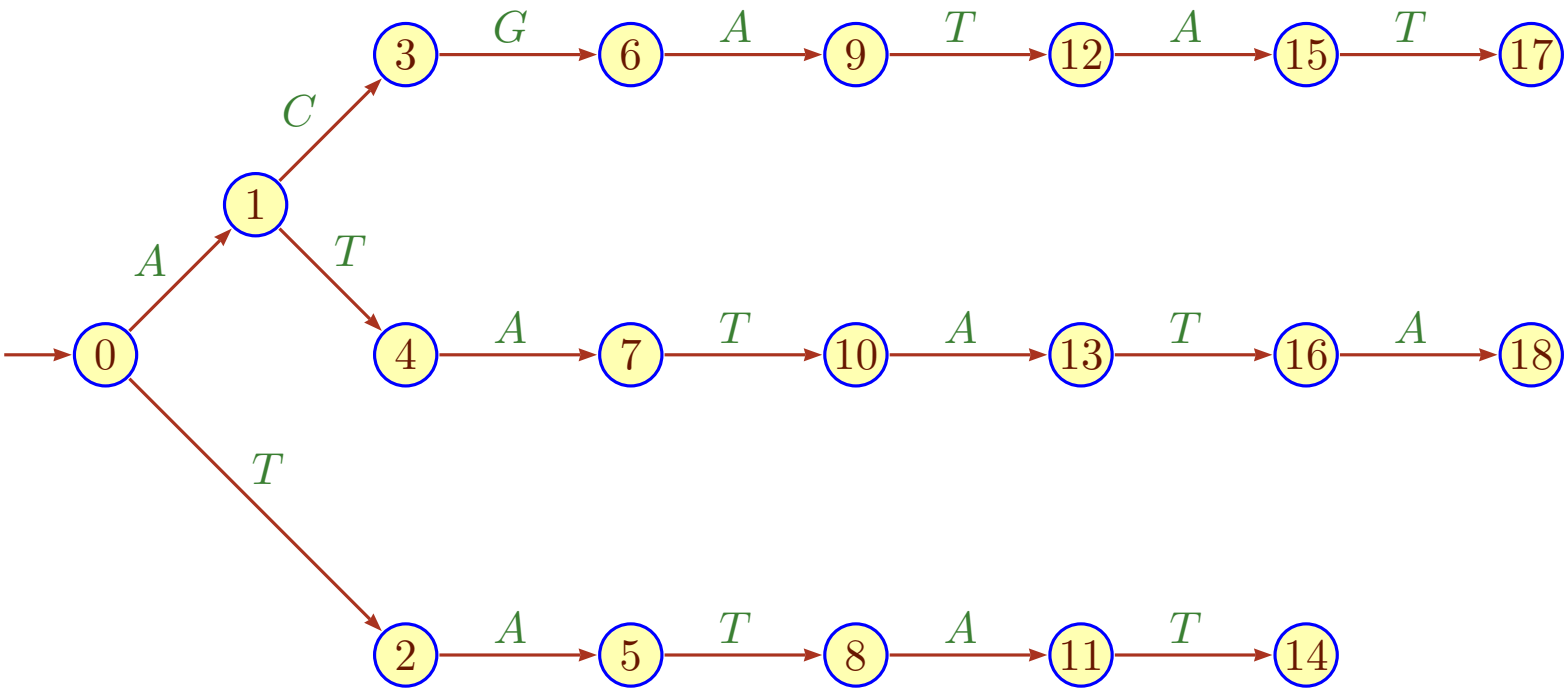
In many situations the real computational problems is not the actual run of the DFA on some input, it's the construction of the DFA in the first place.

In fact, one often avoids the construction of a DFA and makes do with an NFA (see below). When the speed of construction becomes mission-critical, one may even produce a device that is almost a finite state machine and could relatively easily be converted into one, but it is cheaper to use the gizmo directly.

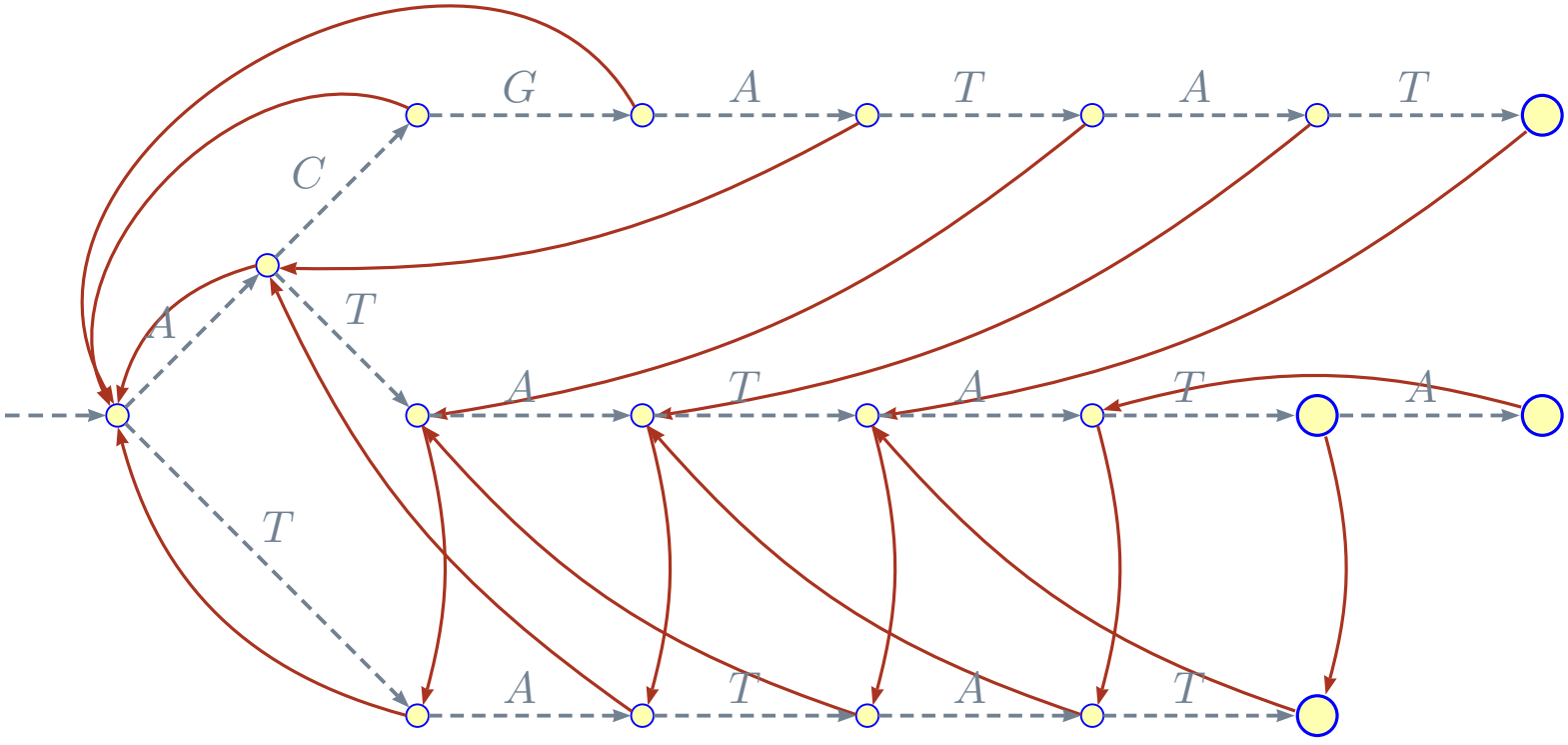
Typical example: string matching.

One is given one or more strings $w_1, \ldots, w_m$ and one needs to find their occurrences in a large text file.

As a good example, consider computational biology, where one looks for particular patterns in an DNA string.



This is the skeleton of a machine that searches for strings ACGATAT, ATATATA and TATAT over alphabet $\{A, C, G, T\}$ (adenin, cytosin, guanin, thymine).



If a mismatch occurs, take a back-transition and then try again. From there it is not hard to construct a proper NFA or, if need be, even a DFA (though efficiency-wise it may be better to stick with slightly more complicated devices).

1 Zero Space

2 Finite State Machines

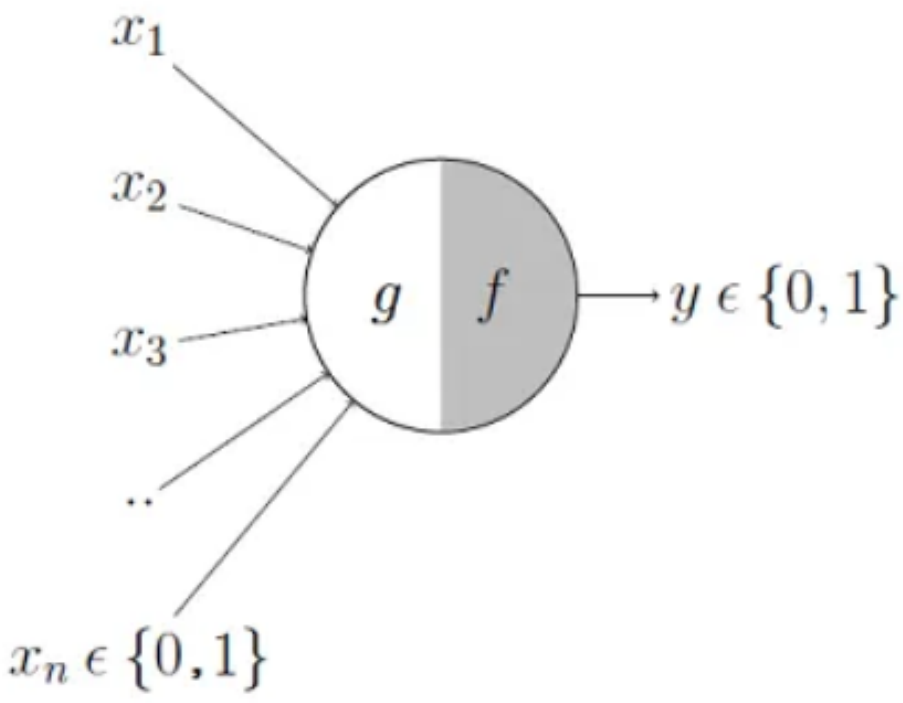
3 **The Foundations**

W. S. McCulloch, W. Pitts
A logical calculus of the ideas immanent in nervous activity
Bull. Math. Biophysics 5 (1943) 115–133

S. C. Kleene
Representation of events in nerve nets and finite automata
in *Automata Studies* (C. Shannon and J. McCarthy, eds.)
Princeton UP, 1956, 3–41.

M. O. Rabin and D. Scott
Finite automata and their decision problems
IBM J. Research and Development, 3 (1959), 114–125.

McCulloch (neuroscientist) and Pitts (logician) present the first attempt to define the functionality of a neuron abstractly. The current AI craze goes back to this paper.



The references in the McCulloch/Pitts paper are rather remarkable.

R. Carnap, The Logical Syntax of Language
Harcourt, Brace and Company 1938.

D. Hilbert, W. Ackermann, Grundzüge der Theoretischen Logik
Springer Verlag 1927.

B. Russell, A. N. Whitehead, Principia Mathematica
Cambridge University Press 1925.

Kleene's paper puts some of the ideas in McCulloch/Pitts on a more solid mathematical foundation and is strikingly elegant. The *nets* under consideration are essentially finite state machines.

- The behavior of a net is a *regular event*, essentially a regular language.
- All regular events can be constructed from trivial ones using simple algebraic operations (synthesis problem, Kleene star).
- Only regular events can be constructed by the algebraic machinery (analysis problem).

The purely algebraic description for regular languages in terms of *regular expressions* is critical in current applications. To wit, if a pattern matching algorithm required a user to type in a finite state machine, it would be essentially unusable. Anyone can type in a regular expression.

The 1959 paper by Rabin and Scott was an absolute breakthrough. For many years it was the most highly cited paper in CS. In particular, it introduced two major ideas:

- nondeterminism in machines,
- decision problems as a tool to study FSMs.

Prior to the paper, computations were always deterministic, the current configuration always determined the next (even though nondeterminism pops up very much by itself in the λ-calculus).

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

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

This sort of transitions makes it possible for computations to branch, the same input may be associated with multiple (in fact, exponentially many) runs.

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

Every language $L \subseteq \Sigma^\star$ presents a natural decision problem: determine whether some word belongs to the language. In the particular case when the language is represented by a FSM we can think of the machine as part of the input (uniform versus non-uniform).

Problem:	**FSM Membership (Recognition)**
Instance:	A FSM $\mathcal{A}$ and a word x.
Question:	Does $\mathcal{A}$ accept input x?

Lemma

The FSM Membership Problem is solvable in linear time.

// recognition problem
$P = I$

while some input symbol $a \in \Sigma$ remains **do**
$P = \{\, q \in Q \mid \exists\, p \in P\, ((p, a, q) \in \tau) \,\}$

return $P \cap F \neq \emptyset$

Note that when the machine is a DFA the state set $P \subseteq Q$ is just a single state, and can be represented by an integer. Recognition is lightning fast in this case.

In general, we need to maintain a container type for P which leads to a modest slow-down in applications. For fixed $\mathcal{A}$, we pick up a multiplicative constant.

It is intuitively clear that DFAs are less complicated than their nondeterministic counterparts. This difference is visible in a minor slow-down in the recognition algorithm for NFAs.

A Challenge:
Can one use other decision problems to distinguish between deterministic and nondeterministic machines?

In particular, are there problems that are, say, polynomial time for DFAs but exponential for NFAs?

There are quite a few natural questions one can ask about FSMs that translate into pretty decision problems.

Problem:	**Emptiness**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept no input?

Problem:	**Finiteness**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept only finitely many inputs?

Problem:	**Universality**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept all inputs?

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 in detail how to deal with Finiteness and Universality.

Theorem

Emptiness and Finiteness for NFAs are decidable in linear time.

The Universality problem for NFAs is PSPACE*-complete.*

In fact, the algorithms for Emptiness and Finiteness are essentially the same as for DFAs (path existence and cycle existence).

The PSPACE-completeness argument is a lot harder, we'll skip.

For any kind of computational model, there is a natural problem called program size complexity: try to find the smallest machine/program in your model that solves a certain problem.

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

For general models of computation such as register machines or Turing machines this problem is not computable. But for FSMs it is more manageable, and for DFAs there is a very good solution.

Note that this approach 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 solver on your wrist watch?

Definition

Two FMSs $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same alphabet are equivalent if they accept the same language: $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$.

So we would like to find the smallest machine in a class of equivalent ones (that all recognize the same language). In some sense, the smallest machine is the best representation of the corresponding language.

Definition

The state complexity of a FSM is the number of its states.
The state complexity of a recognizable language L is the size of a smallest DFA accepting L.

We wind up with another decision problem (this is really an optimization problem, but we can express in the usual slightly twisted form):

Problem:	**State Complexity**
Instance:	A recognizable language L, a bound β.
Solution:	Is the state complexity of L at most β?

Again, there is a gap between deterministic and nondeterministic machines.

Theorem

State Complexity is polynomial time for DFAs, but PSPACE*-complete for NFAs.*

The obvious brute-force algorithm for state complexity is to

- generate all DFAs in order of increasing size, and
- check all of them for equivalence with the given machine;
- stop when the first match pops up.

There are a few problems with this method. First, we need to make sure that every recognizable language is already accepted by a DFA. Second, the search may involve exponentially many DFAs. Third, we don't know how to check whether two machines are equivalent.

We will postpone the first problem, ignore the second and only handle the third one.

Lemma

Equivalence of finite state machines is decidable.

Proof.

It suffices to find some bound β so that $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent iff they agree on all words of length β. We may safely assume that both machines have n states. Consider some word x of length m that the machines disagree on.

We trace the computations of both machines on x. Write $x_{\leq i}$ for the prefix of x of length i.

$$S_i = \big(\tau_1(I_1, x_{\leq i}), \tau_2(I_2, x_{\leq i})\big)$$

If $m \geq (2^n)^2$, then for some $0 \leq i < j \leq m$ we must have $S_i = S_j$. But then we can shorten x by removing the factor $x_{i+1}, \ldots, x_j$. Repeating this sort of surgery ultimately produces a string of length at most $\beta = 4^n$ works.

□

Note that For DFAs the bound can immediately be improved to $\beta = n^2$.

Great, but still totally useless since we have to check k^β words where $k = |\Sigma|$.

The real challenge is to try to understand how efficient we can make equivalence testing, and how efficiently we can determine state complexity of a regular language.

Finding good algorithms requires a much better understanding of FSMs.