CDM

Memoryless Machines

Klaus Sutner

Carnegie Mellon University

Spring 2025

1 Zero Space

2 Finite State Machines

3 The Foundations

Getting Very Real

We have seen two model of computation: Turing machines and primitive recursive functions. Both of them blissfully ignore physical limitations, the wild computations we talked about (Ackermann, Busy Goodstein, Beaver) demonstrate clearly that a lot of computations are in now way physically realizable, even though the underlying logic is not particularly complicated.

How about exploring the bottom end of computability?

To simplify matters, we will focus on decision problems (acceptors) and postpone function problems (transducers).

Decision Problems

For us, a decision problem Π consists of

- a set of instances I_Π
- a set of Yes-instances $Y_\Pi \subseteq I_\Pi$

In other words, we are interested in recognizing instances that have some special property:

$$Y_\Pi = \{\, x \in I_\Pi \mid P(x) \,\}$$

It is customary to specify problems in the form

Problem:	**RiddleMeThis (RMT)**
Instance:	Some instance x.
Question:	Does x have property P?

Keeping Things Infinite

Constraint:
We will only consider infinite decision problems:
the set of instances must be infinite.

Typical examples of sets of instances are (subsets of) $\mathbb{N}$, $\mathbf{2}^\star$ or $\Sigma^\star$.

To be clear, the set of Yes-instances may very well be finite, or even empty.

Digression: Instances

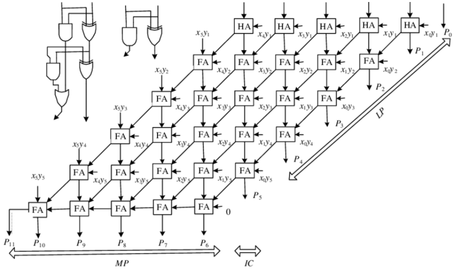
In concrete problems we encode some combinatorial structure as a string, say, in $\mathbf{2}^\star$. Usually not every string is an instance. E.g., a directed graph on n points may be represented by a binary string of length n^2.

We will ignore this issue and merrily pretend that $\mathbf{2}^\star$ is the set of instances.

This is justified by the fact that, for any reasonable encoding, it is trivial to check whether a string is an instance. If not, we simply return No.

Finite is Important, Too

This is not to say that algorithms that operation on some fixed, finite number of bits are hugely important. E.g., this is a circuit for parallel multiplication.



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

Why Infinite?

Claim:

Every finite decision problem is decidable in constant time.

This is entirely correct, albeit for entirely the wrong reasons.

We can simply hardwire a lookup table that lists the correct answer for each instance. Since we can always order the instances in some natural way, this requires essentially just a bitvector of length $|I|$.

Easy.

Problem 1

We can push this to absurd levels:

Problem:	**Riemann Hypothesis (RH)**
Instance:	A banana.
Question:	Is the Riemann hypothesis true?

If you don't like bananas, use a beer-mug instead.
This problem is easily decidable.

Algorithm I: Eat the banana, return Yes.
Algorithm II: Eat the banana, return No.

One of those two algorithms works. Of course, that is useless, we need a constructive solution not some abstract existence proof[†].

[†] This is one occasion where classical logic does not work well.

Problem 2

Even if we could compute the entries in the lookup table, there is still the problem of physical realizability.

E.g., think about multiplying two 64-bit numbers. There are $2^{128} \approx 3.4 \times 10^{38}$ entries, each 64 bits long. So the whole table takes

$$2^{134} \approx 2.2 \times 10^{40}$$

bits. Utterly unmanageable.

StringWorld$^{\text{TM}}$

At this point we should also give up on arithmetic.

In principle we can use sequence numbers to code any conceivable finitary structure, but that requires overhead that obscures the details of simple computations. If we want to explore the bottom level of computability we cannot afford all the encoding and decoding.

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 already have a perfect model of computation.

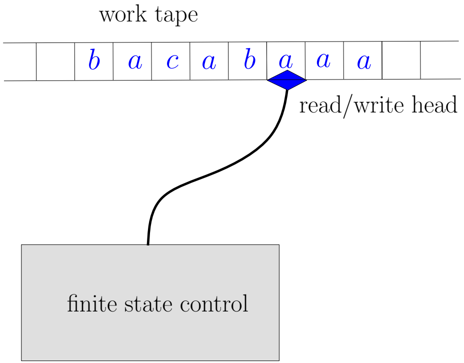
Recall: PR String Functions

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.

We could try to work with these functions by, say, limiting the number of nested primitive recursions needed to define an operation.

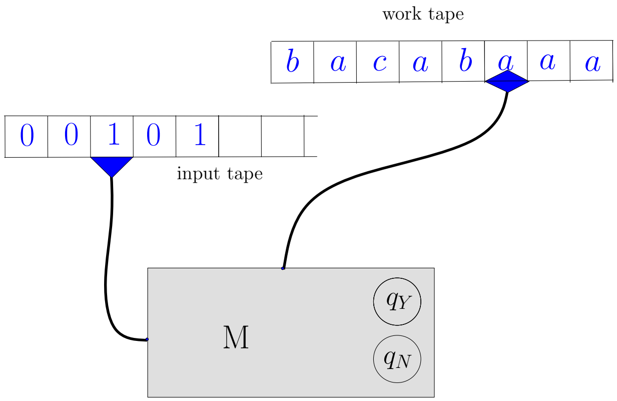
This is fine in principle, but we want an even more fine-grained approach. To this end we will start with Turing machines and add a number of restrictions.

Turing Machines, Again



In the standard model, input is written on the worktape.
This makes it a bit complicated to talk about the space complexity of the machine: the input really should not count, just the extra space needed during the computation.

Separate Input



A modified machine with separate input tape. This is an acceptor signaling the answer by two special halting states.

Here we can limit the amount of space the machine can use on the worktape.

Space Complexity

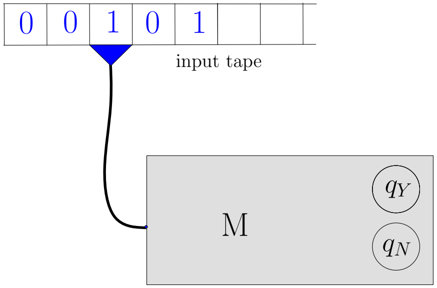
Suppose $\mathcal{M}$ is a TM with separate input tape.
The space complexity $S : \mathbb{N} \to \mathbb{N}$ of $\mathcal{M}$ is defined as

$$S(n) = \text{max workspace used in any computation on } x \in \Sigma^n$$

Popular space complexities are n and $\log n$.

Sublinear complexity for space is quite interesting (but not for time).

Zero Space



Here is radical proposal: Suppose we allow no worktape whatsoever.

The machine can read the input and perform state transitions, but there is no scratch-space whatsoever.

Constant is Zero

If you feel existential angst about zero space, suppose we fix the worktape to have some constant length m.

This is no different from zero space.

There are $m\,|\Gamma|^m$ many possible worktape contents and head positions. We simply make them part of the state set:

$$Q' = Q \times [m] \times \Gamma^m$$

Note the exponential blowup in the size of the machine, though.

Little Space

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

Here is a major surprise:

Theorem (Hartmanis, Lewis, Stearns 1965)

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

Hence, once we get to $S(n) = o(\log\log n)$ we might as well have a worktape of fixed size. The proof is interesting, see Hartmanis Eal.

Left-To-Right

The head on the input tape can move left and right.

As it turns out, we may assume that the read head only moves from left to right: at each step one symbol is scanned and then the head moves right and never returns.

Theorem (Rabin/Scott, Shepherdson 1959)

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

The proof of this result is quite messy, we won't go into details. Unsurprisingly, the one-way version has more internal states. See Rabin/Scott 1959.

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 odd?
- **Majority:** Are there more 1-bits than 0-bits in x?

Parity can easily be handled without memory: keep one parity bit (initially 0), then read the input and flip the bit whenever you see a 1.

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

Parity Checker

```
// parity checker
    $p = 0$
    while $a = x$.next() do
        $p = p \oplus a$
    return $p$
```

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

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

Digression: Streaming Algorithms

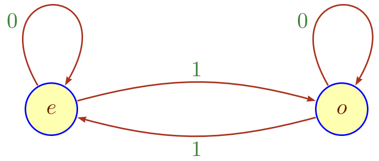
This is an extremely simple case of a streaming algorithm: these us a small number of scans and little memory (typically logarithmic).

```
initialize
while $a = x$.next() do
    process $a$
maybe repeat
return answer
```

In the era of big data this sort of algorithm is very important.

Transition Diagrams

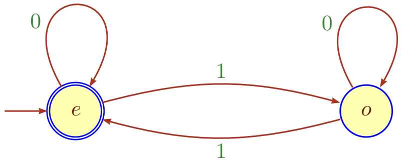
A most useful representation for our parity checker is a diagram:


$$\begin{tikzcd} e \arrow[loop above, "0"] \arrow[r, bend left, "1"] & o \arrow[loop above, "0"] \arrow[l, bend left, "1"] \end{tikzcd}$$

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

Complete Information

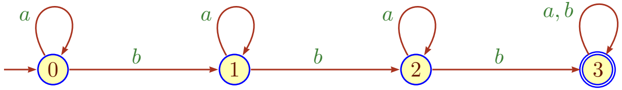
It is customary to indicate the initial state (where all computations start) by a sourceless arrow, and the so-called final states states (corresponding to answer Yes) by marking the nodes.



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

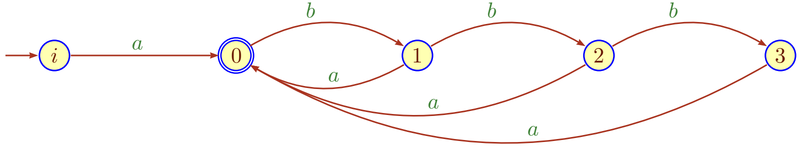

$$\begin{tikzcd}
\arrow[r] & 0 \arrow[r, "b"] \arrow[loop above, "a"] & 1 \arrow[r, "b"] \arrow[loop above, "a"] & 2 \arrow[r, "b"] \arrow[loop above, "a"] & 3 \arrow[loop above, "{a,b}"]
\end{tikzcd}$$

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.

Run-Length Limited Codes

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.



This machine has missing transitions: reading a b in state i “crashes” the computation. As a practical matter, partial transitions are critical for efficiency.

A detail: our informal description does not explain whether input a is allowed.

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 (actually, one checks the first 100 or so primes).

Assume n has 1000 bits. Using a 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?

Induction to the Rescue

Write $\mathsf{val}(x)$ for the numerical value of bit-sequence x, assume MSD first.

Then

$$\mathsf{val}(x0) = 2 \cdot \mathsf{val}(x)$$

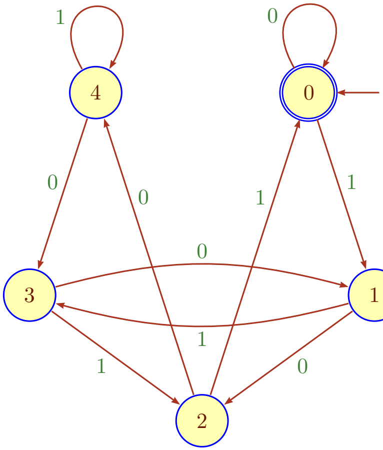
$$\mathsf{val}(x1) = 2 \cdot \mathsf{val}(x) + 1$$

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

$$\mathsf{val}(xa) = 2 \cdot \mathsf{val}(x) + a \pmod 5$$

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.

Remainders Mod 5



Optimality in Time

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

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

Then this algorithm cannot look at all the bits in the input, so it will not notice a single bit change 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 $k \geq 0$.

1 Zero Space

2 Finite State Machines

3 The Foundations

The Machine Perspective

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 $p \xrightarrow{a} q$.

Sequences, Words, Strings

Given an alphabet Σ one writes

$\Sigma^\star$	the collection of all words over Σ
Σ^+	the collection of all non-empty words

We allow any finite, non-empty set as an alphabet. 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.

Finite State Machines

Definition

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

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

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

The acceptance condition determines whether an FA accepts or recognizes some input $x \in \Sigma^\star$. We won't try to give a general definition and simply explain various examples as we go along.

The (acceptance) language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\star$ of the automaton $\mathcal{A}$ is

$$\mathcal{L}(\mathcal{A}) = \{\, x \in \Sigma^\star \mid \mathcal{A} \text{ accepts } x \,\}$$

Aka the language recognized by $\mathcal{A}$.

Vanilla Acceptance

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

Vanilla acceptance:

Some computation on input x starts in a state in I and ends in a state in F.

As before for Turing machines, the output/answer depends only on the last configuration (intermediate steps do not matter).

Configurations

Since finite state machines are simplified Turing machines we can use our old configuration/one-step approach to define computation. Configurations are simple: we only need the current state $p \in Q$ and the remainder $z \in \Sigma^\star$ of the input.

$$\mathcal{C} = \{\, p\,z \mid p \in Q, z \in \Sigma^\star \,\}$$

One step in a computation is then given by the lookup table τ.

$$p\,az \mathrel{\Big|\!\frac{1}{\mathcal{A}}}\, q\,z \iff \tau(p, a, q)$$

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

Note that this time we are dealing with a relation, not a function in general.

Accepting Computations

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:

$$p\,x \mathrel{\vdash^{*}_{\mathcal{A}}} q \qquad p \in I, q \in F$$

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.

What is decent algorithm to test acceptance?

Runs

Fix some transition system $\mathcal{A} = \langle Q, \Sigma, \tau \rangle$. Given a word $u = u_1 u_2 \ldots u_m$ over Σ, a run of $\mathcal{A}$ on u is an alternating sequence of states and letters

$$\pi = p_0, u_1, p_1, u_2, p_2, \ldots, p_{m-1}, u_m, p_m$$

such that $p_{i-1} \xrightarrow{u_i} p_i$ is a valid transition for all i. So a run is just a path in a labeled digraph.

p_0 is the source of the run and p_m its target, and $m \geq 0$ its length. A run is accepting if its source is in I and its target in F.

Occasionally we abuse terminology and refer to the corresponding sequence of states alone as a run:

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

Traces

Given a run

$$\pi = p_0, u_1, p_1, u_2, p_2, \ldots, p_{m-1}, u_m, p_m$$

of an automaton, the corresponding sequence of labels

$$u = u_1 u_2 \ldots u_{m-1} u_m \in \Sigma^\star$$

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.

Acceptance Testing

We can rephrase the definition of the acceptance language like so;

$$\mathcal{L}(\mathcal{A}) = \{\, x \in \Sigma^\star \mid \mathcal{A} \text{ has an accepting run with trace } x \,\}$$

Hence, testing whether a FSM accepts some string comes down to a particular path existence problem in the transition system. This is algorithmically easy, and naturally leads to lighting fast methods that work well on very long inputs (think big data or computational biology).

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
coaccessible	if F is reachable from it
trim	if it is both accessible and coaccessible
trap	if all transitions with source p have target p
sink	if it is a trap but not final

One uses similar terminology for the whole automaton: all states are accessible/coaccessible/trim.

By removing states that don't have the requisite property we obtain the accessible/coaccessible/trim part of a machine.

Cleaning Up

To compute the accessible/coaccessible/trim part of a machine we can use standard graph exploration algorithms such as DFS or BFS: these are all reachability problems.

Importantly, all these parts can be computed in linear time and space.

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

Alas, building a large machine and then filtering out the trim part is counterproductive, preferably one should try to construct only accessible machines in the first place (trim is tricky).

Civilized Transitions

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 $a \in \Sigma$

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

A deterministic system has at most one run from a given state for any input.

In Other Words

A deterministic transition system consists of a partial function, the transition function

$$\delta : Q \times \Sigma \nrightarrow Q$$

By currying, we can also think of a family of (symbol) transition maps

$$\delta_a : Q \nrightarrow Q$$

where $a \in \Sigma$.

If the transition system is in addition complete, we get plain functions

$$\delta : Q \times \Sigma \to Q \qquad \delta_a : Q \to Q$$

DFAs

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 speak of a deterministic finite automaton (DFA). We use the vanilla acceptance condition (path from q_0 to F).

It is straightforward to see that given $x \in \Sigma^\star$:

- a PDFA has at most one run starting at q_0 with trace x
- a DFA has exactly one run starting at q_0 with trace x

Comments

Arguably, DFAs should be called *complete, deterministic finite automata*, acronym CDFA. Unfortunately, that ship has sailed . . .

We 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.

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

Recognizable Languages

Definition

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

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.

Fast Acceptance Testing

Proposition

For any PDFA $\mathcal{A}$ and any input string x we can test in time linear in $|x|$ whether $\mathcal{A}$ accepts x, with very small constants.

```
$p = q_0$
while $a = x$.next() do
    $p = \delta(p, a)$
return $p \in F$
```

Here it is understood that the right thing happens when $\delta(p, a) \uparrow$.

Weirdness

Note that we are using a slightly strange approach here: usually one first defines a class of functions (TM 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.

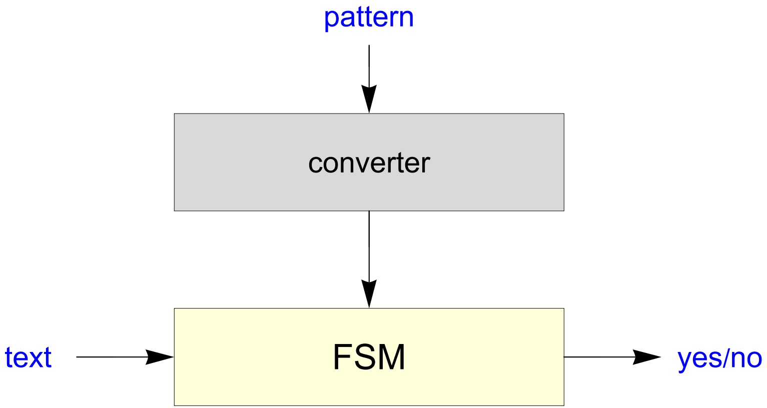
The Killer Apps

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`, `emacs` or `rg`). All compilers use similar tools.

2. There is a 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.

Pattern Matching



Truth in Advertising

In RealWorld$^{\text{TM}}$ situations one often uses algorithms that are based on finite state machine concepts, but use additional hacks to speed things up or provide additional power. For example, regular expressions can often express more than just regular languages.

Our emphasis on PDFAs so far is misleading, one often avoids the construction of a DFA and makes do with an NFA instead: a nondeterministic machine may be exponentially smaller than its deterministic counterpart, but still work fine (e.g. for pattren matching).

1 Zero Space

2 Finite State Machines

3 The Foundations

The Early Days

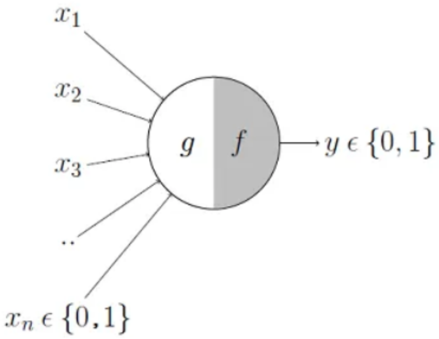
W. S. McCulloch, W. Pitts
A logical calculus of the ideas immanent in nervous activity
Bull. Math. Biophysics 5 (1943) 115–133

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.

Neural Nets

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.



References

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

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 Breakthrough

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).

Nondeterminism

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.

The Membership Problem

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

Lemma

The FSM Recognition Problem is solvable in linear time.

Killer App

```
// recognition problem
$P = I$
**while** $a = x$.next() **do**
    $P = \{\, q \in Q \mid \exists\, p \in P\, \tau(p, a, q) \,\}$

**return** $P \cap F \neq \emptyset$
```

When the machine is a PDFA we can replace $P \subseteq Q$ by an integer.

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. In practice, the constant is often small.

Killer Idea

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?

More Decision Problems

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?

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 (remove edge labels).

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).

□

Equivalently, the trim part of the machine is empty.

Not so Easy

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.

Anther Killer Idea

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?

Equivalence and State Complexity

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.

Determining State Complexity

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?

The language L is supposed to be given as a finite state machine. Again, there is a gap between deterministic and nondeterministic machines.

Theorem

State Complexity is polynomial time for DFAs.
It is PSPACE*-complete for NFAs in general.*