CS311 Computational Structures

Turing Machine Variants

Lecture 12

Andrew Black
Andrew Tolmach

The Church-Turing Thesis

The problems that can be decided by an algorithm are exactly those that can be decided by a Turing machine.

The Church-Turing Thesis

- The Church-Turing cannot be proved; it is essentially a formal **definition** of the word “algorithm.”

- However, there’s lots of evidence that our intuitive notion of algorithm is equivalent to a TM, and

- no convincing counter-examples have yet been found.

Evidence: Robustness

- The Turing machine is remarkably robust
  - Tweaking the definition, even in major ways, does not affect what the Turing machine can do.

Variant Turing Machines

- There are many possible variations on the definition of a Turing machine.
  - Add heads, tapes, RAM, non-determinism, etc.
  - Restrict motion on tapes, symbols, etc.
- But they all have equivalent power!
  - More precisely, they recognize (or decide) the same class of languages.
  - Assumption: can do only finite work at each step

Simulations

- To show that two kinds of machines are equivalent, we show how to **simulate** the behavior of one kind using the other.
  - More specifically: given a machine M of one kind that recognizes a language L, show how to construct a machine M' of the other kind that recognizes L, and vice-versa.
  - “bi-simulation”
- We ignore efficiency concerns (for now)
  - Take advantage of large (though finite) sets of states and tape alphabets

Some equivalent extended TMs

- Multiple heads
- **Multiple tapes**
- Two-dimensional tapes
- **Random-access memory TM's**
- **Non-deterministic TM's**

Adding Multiple Tapes

- Consider a TM variant where we have *k* tapes each with its own independent head
  - Transition function has the form:
    $\delta(q,a_1,...,a_k) = (q',b_1,...,b_k,L/R/S_1,...,L/R/S_k)$
  - Start with input on tape 1; other tapes blank
- We can simulate a machine of this kind using an ordinary TM by conceptually dividing the TM tape into multiple **tracks**
  - Each tape slot contains an *n*-tuple of symbols; this is just a way of thinking about a (large!) alphabet

Simulating k tapes with $2k$ tracks

- We use two tracks to simulate each tape:
  - One track contains symbols of the tape
  - The other contains a 1 at the spot representing the position of the tape head, and 0s elsewhere

Tape1 {	a	b	a	a	#	b	a			
	0	0	0	1	0	0	0	0	0	0
Tape2 {			a	#	b	a	a	#		
	0	0	1	0	0	0	0	0	0	0
Tape3 {		a	a	a	b	b	b	b	b	a
	0	0	0	0	0	0	0	1	0	0

...

Simulating k tapes with $2k$ tracks

- Why don't we store the head positions in the finite control?

Simulation scheme

1. Initialize tape by changing input $w_1w_2...w_n$ to $[w_1,1, \sqcup,1,..., \sqcup,1][w_2,0, \sqcup,0,..., \sqcup,0]...[w_n,0, \sqcup,0,..., \sqcup,0]$

2. To simulate a step of M:

   a. scan entire live portion of tape to determine which symbol is under each head; store in finite control

   b. use M's δ to decide new state and how each tape should change and each head should move

   c. scan over tape again to make these changes

3. Halt with accept/reject when M would

Adding Random Access

- TM's may seem impossibly less powerful than real computers
  - Despite their having the super-power of storing an indefinitely large amount of information on tape!
- One thing that real machines have is an addressable memory
- Can we extend the TM model to include such a memory?
  - Indeed, a super-memory of indefinitely large size

Simulating Random Access

- Such a RAM TM is easy to simulate with a multi-tape TM
- How?
  - Note: we don't need the simulation to be **efficient**!

Non-deterministic TMs

- We can add non-determinism to TMs in the usual way, by extending the transition function to $\delta : Q \times \Gamma \rightarrow \wp\ (Q \times \Gamma \times \{L,R\})$
- Very handy for “guess and test” style algorithms
- As with FA’s (and unlike with PDA’s), adding non-determinism makes no difference to the languages that can be recognized

Simulating non-determinism

- Use a 3-tape TM
  - Tape 1: input string (treated as read-only)
  - Tape 2: contents of NTM tape
  - Tape 3: control sequence (a.k.a. “address tape”)
- Basic idea: TM tries every possible combination of guesses made by NTM
  - Control sequence describes one combination of guesses. After each trial, it is updated to describe a new combination of guesses.
- If NTM accepts a string, TM will eventually try a choice that accepts it

Control sequence details

- Suppose that the number of possible alternatives for any NTM transition is ≤ b.
- Then we can code each guess as a number in {1,...,b} and each collection of guesses as a sequence {1,..,b}*.
  - Example: “213” means “at the first ND guess choose the 2nd alternative, at the next one choose the 1st alternative, at the last one choose the 3rd alt.”
  - If the control sequence asks for choice k at a point where there are fewer than k choices, reject on this trial.
- To guarantee that we try **all** such sequences, we generate them in **lexicographic** order:
  - 1,2,...,b,11,12,...,1b,21,22,...,2b,...,b1,b2,...,bb,111,...

Simulating language recognition

- Start with tape 1: input, tape 2: empty, tape 3: control sequence 1.
- Loop performing trials until acceptance:
  - Copy input to tape 2
  - Simulate behavior of NTM using tape 3 to control guesses.
    - If control string is exhausted, reject on this trial
  - If enter an accept state, stop and accept.
  - If enter a reject state, reject on this trial
  - Update tape 3 with next sequence; clear tape 2

Some Equivalent Restricted TMs

- TM with semi-infinite tape (see IALC)
- **k-PDA's for k ≥ 2.**
- **k-counter machines for k ≥ 2.**

k-PDA's

- A *k*-PDA is just like a PDA but with *k* stacks.
  - Transition function has the form δ:
    $Q \times \Sigma \times \Gamma_1 \times \Gamma_2 \times ... \times \Gamma_k \rightarrow$
    $\wp\ (Q \times \Gamma_1 \times \Gamma_2 \times ... \times \Gamma_k)$
  - 1-PDA is an ordinary (N)PDA
  - 0-PDA is just a NFA
- We know that a TM is more powerful than a 1-PDA. Evidence?

2-PDA's

- It turns out that a 2-PDA can simulate a Turing machine.
  - How?
    - Hint: it's all a question of how to represent the tape; everything else is easy.
- Hence, a 2-PDA can simulate a k-PDA for any $k > 2$.
  - Easy to see that a TM can simulate a k-PDA

Counter machines

- A counter is a register that we can (only)
  - increment
  - decrement (if non-zero)
  - test for zero

- It is equivalent to a stack with $\Gamma = \{Z_0, X\}$ with contents always of the form $X^n Z_0$
- A *k*-counter machine consists of a finite state control, a read-only input tape, and *k* counters

3-counter machine simulates TM

- Will show 3-counter machine simulates 2-PDA
- Suppose 2-PDA stack alphabet is $\{Z_1, Z_2, \ldots, Z_{k-1}\}$
- We'll represent the stack $Z_{i0}Z_{i1}\ldots Z_{im}$ by the integer $j = i_0 + i_1 k + i_2 k^2 + \ldots + i_m k^m$.
- Use one counter to represent each stack; use the third counter for scratch work.
- Must show we can simulate pushing, popping, and reading the top of stack.

3-counter: simulating push

- Recall $Z_{i0}Z_{i1}...Z_{im} \sim j = i_0k^0 + i_1k^1 + ... + i_mk^m$
- To **push** Z_r, must change j to jk + r
  - Zero scratch counter
  - Repeatedly decrement stack counter and increment scratch counter by k, until stack counter is 0
    - can use finite control to count up to k.
  - Move scratch counter to stack counter.
  - Increment stack counter by r (using finite control)

3-counter: simulating pop

- Recall $Z_{i0}Z_{i1}...Z_{im} \sim j = i_0k^0 + i_1k^1 + ... + i_mk^m$
- To **pop** Z_{i0}, must change j to j/k (ignoring remainder)
  - Zero scratch counter
  - Repeatedly decrement stack counter by k and increment scratch counter by 1, until stack counter is < k
    - can use finite control to count up to k.
  - Move scratch counter to stack counter.

3-counter: simulating read

- Recall $Z_{i0}Z_{i1}...Z_{im} \sim j = i_0k^0 + i_1k^1 + ... + i_mk^m$
- To **read** Z_{i0}, must compute j mod k.
  - Zero scratch counter
  - Copy stack counter to scratch counter keeping track of its value mod k in the finite control
    - generalizes DFA tracking odd/even count
  - Copy scratch counter back to stack counter

2-counter machine simulates TM

- Will show 2-counter machine with counters A,B can simulate a 3-counter machine with counters P,Q,R.
- Idea: Represent the state of the 3-counter machine with P=i, Q=j, R=k by the single value $p^i q^j r^k$ where p,q,r are any three distinct prime numbers
- Store this value in A; use B for scratch.
- To simulate incrementing counter P, multiply A by p
  - Already saw how to do this with one scratch reg.
  - Similarly for Q,R.
- To simulate decrementing counter P, divide A by p
- To simulate checking if counter P is non-zero, check if A is evenly divisible by p

Big Idea Coming up

Big Idea Coming up

Universal Turing Machines

A universal Turing Machine:

1. takes as input a description of some other Turning machine M, and
2. an input string w, and
3. simulates the action of M running on w,
4. halting, looping or accepting as does M.

How to build a Universal TM

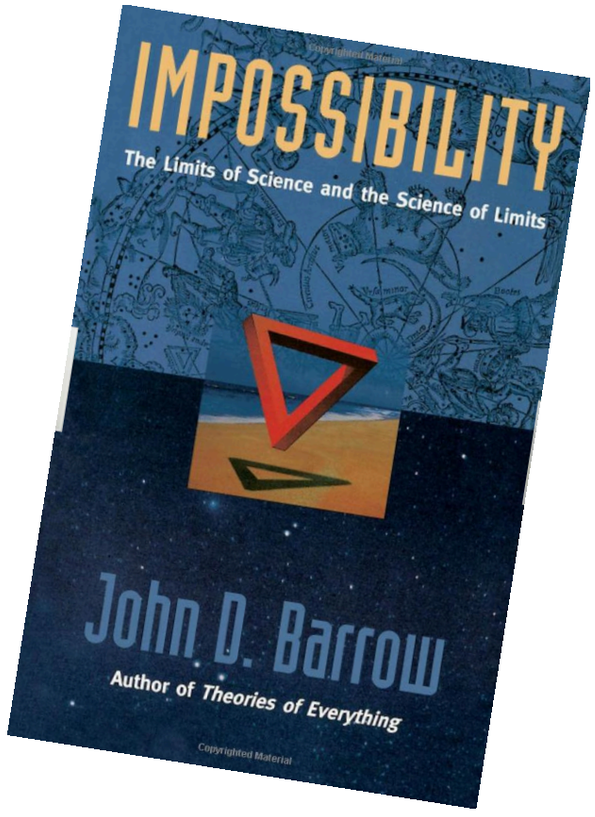
- Remember, a Universal TM U must still have only a fixed and *finite* set of states, a *finite* alphabet, and therefore a *finite* set of instructions (δ).
- Relabel the TM that we are simulating, S, to use states from $\mathbb{N}$ and symbols from $\{a_i \mid i \in \mathbb{N}\}$.
  - That's OK, there are bound to be enough!
  - Pick a fixed alphabet for U, say $\Gamma_U = \{0,1,..,\}$
  - *Encode* the instructions of S in Γ_U

- Let U have three tapes:
  - tape 1 stores the encoding of S
  - tape 2 stores the input string
  - tape 3 records the state of S, encoded in Γ_U
- U does the following:
  - Writes the start state on tape 3
  - If the state on tape 3 is a final state, then halt
  - otherwise, read the current symbol from tape 2 and find the instruction on tape 1 that applies.
  - Write the new state onto tape 3, and perform the indicated write and move operations on tape 2.

Why are UTMs important?

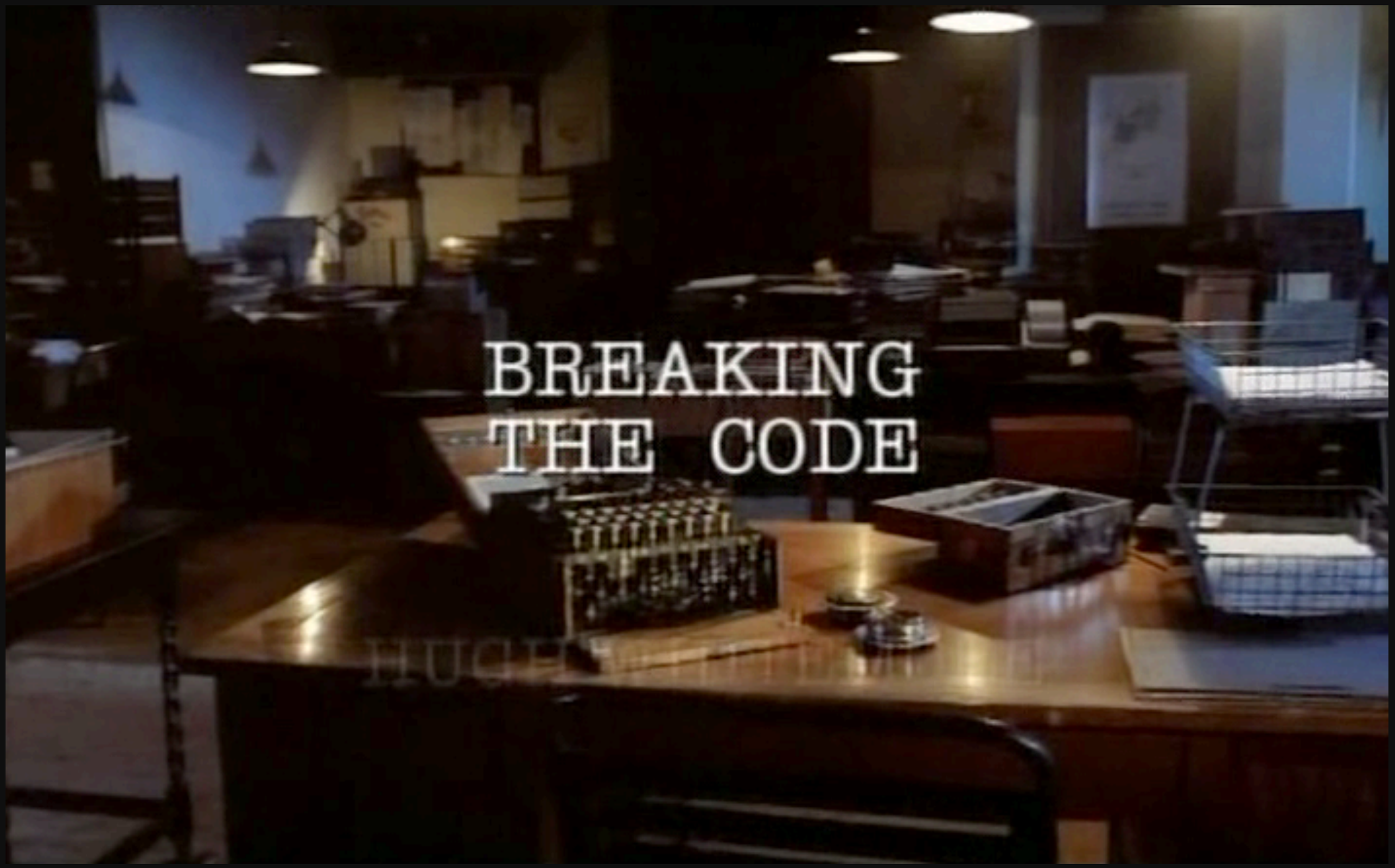
- They capture the idea of a stored-program computer
  - First stored-program computer, Manchester "Baby" was built in 1948
    - Earlier computers, Colossus, Harvard Mk I and ENIAC, were fixed program computers.
- The idea of a computer that can take another computer as input is key to proofs of undecidability and unsolvability.

From: *Impossibility*, by John Barrow.
Oxford 1998



There is an intriguing pattern to many areas of deep human inquiry. Observations of the world are made; patterns are discerned and described by mathematical formulae. The formulae predict more and more of what is seen, and our confidence in their explanatory and predictive power grows. Over a long period of time the formulae seem to be infallible: everything they predict is seen. Users of the magic formulae begin to argue that they will allow us to understand everything. The end of some branch of human inquiry seems to be in sight. Books start to be written, prizes begin to be awarded, and of the giving of popular expositions there is no end. But then something unexpected happens. It's not that the formulae are contradicted by Nature. It's not that something is seen which takes the formulae by surprise. Something much more unusual happens. The formulae fall victim of a form of civil war: they predict that there are things which they cannot predict, observations which cannot be made, statements whose truth they can neither affirm nor deny. The theory proves to be limited, not merely in its sphere of applicability, but to be *self*-limiting. Without ever revealing an internal inconsistency, or failing to account for something we have seen in the world, the theory produces a 'no-go' statement. We shall see that only unrealistically simple scientific theories avoid

this fate. Logical descriptions of complex worlds contain within themselves the seeds of their own limitation. A world that was simple enough to be fully known would be too simple to contain conscious observers who might know it.


BREAKING
THE CODE

Dilly Knox meets Alan Turing


Dilly Knox meets Alan Turing


DIVX
VIDEO