CS1003/1004: Intro to CS, Spring 2004

Lecture #14: AI, Computation Theory, The End

Janak J Parekh
janak@cs.columbia.edu

Administrivia

- HW#6 due next Wednesday – work on it!
- Maryam out this week
  - William will be teaching her lectures and covering Thursday's OH
  - We'll cover her OH next Monday as well
- OH requests next week?
- Review sessions – Tues. and Thurs.
  - Room to be finalized; will send email
  - Got preferences?
- Grades are up, please check them out ASAP

AI continued: Robotics/vision

- Historically focused on mechanical and electrical engineering aspects
- We can already do set tasks, but what about modifications?
  - Objects on a conveyor belt at irregular intervals/orientation
  - Navigate around a room with obstructions
- Need to take images of scenes, compute boundaries, detemine paths
- Goal: autonomous robots

Database/expert systems

- Context drives a huge problem: how to encode context and knowledge that the human mind possesses, and retrieve said information?
- "Associative memory systems"
- Web search is just a start – just keyword-based searching so far, not semantic-based searching
- Expert systems: encode domain-specific knowledge to help solve problems

Weak vs. Strong AI

- All of these applications are essentially *weak*: we tell the computer what to do, and we solve problems
  - Not really "AI", per se – useful solutions to solve real-world problems
- *Is* Strong AI, i.e., sentience/consciousness, possible?
  - If so, we're still quite a long way away
  - On the other hand, there's the Turing test…

So… what can't computers do?

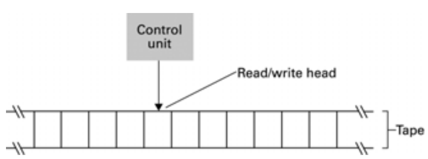
- (Or, can we summarize what *can* they do?)
- Given all that we've learned this semester, it's actually pretty hard to characterize
- Focus of *computation theory* is to determine what is computable and what is not
  - Computable implies functions whose output values can be determined algorithmically from their input values
  - So, what's an example of a noncomputable function?

Formalizing computability

- Several popular ways
  - (Finite) state machines
  - Turing machines
- State machines are a sort of like a flowchart
  - One starts at a "start state", goal is to get to the "end" or "goal" state
  - State *transitions* specify what to do based on initial input
  - States represent the "current" computer's state
  - Simple example: build a state machine to match the string "Hello!"
  - Problem: intermediate storage?

Turing machine

- A state machine on steroids
- Idea: not only do we have *state*, but we have *storage*
- Alan Turing modeled the storage as a "paper tape" in 1936
- The tape is manipulated by a read/write head that can move left and right one space



Simple Turing example

- Add one to a number already encoded on tape
- We encode it as a binary number, and surround it with the start/end states ("*")
- Let's do this on the board…

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

So why bother with Turing?

- Church-Turing thesis: the set of Turing functions is the *same* as the set of functions that are computable in general!
  - Although some may look *really* awkward in a Turing machine
- Widely accepted by computer scientists today
- A language is *Turing-complete* if it can encode all that a Turing machine can do
  - Both C and Java are Turing-complete

Noncomputability, redux

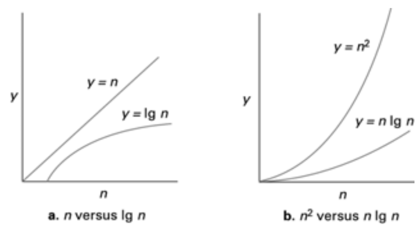
- So, noncomputable functions can't be modeled as a Turing machine
- How do we demonstrate?
  - Not that trivial, beyond scope of class
- Most famous noncomputable function: *Will a specified program halt?*

The "halting problem"

- In short, we *cannot* compute whether or not a computer program written in a Turing-complete programming language will run to completion or not!
  - Note that the *program itself* is "input" into this noncomputable function (e.g., willHalt(…))
- Informal proof is in book; strictly optional (but you may find it interesting)
  - Bare-Bones also optional

Classes of computable functions

- We typically break them down by the time they take to run; here are some typical values that we've seen:

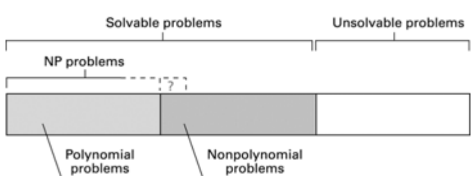


"Bad" computable functions

- Those that, for *any implementation*, take exponential time
- For sufficient *n*, these problems take so long to run that no matter how fast your computer is, it'll still take practically forever
- What's scary, though, is that there is (currently) *no way of proving* that there is *no faster way* of computing it
  - While recursive Fibonacci is bad, iterative is not

So…

- We call such functions for which we know no better way to be "nondeterministic polynomial", or NP
  - Typically exponential
- We care because lots of useful problems fall into this category



How does one "prove" NP?

- You show that one non-polynomial problem reduces into another non-polynomial problem
  - **NP-complete** problem
  - Can't do for all NP problems, but for many of them
  - It's a "weak" proof: if one were to demonstrate that there exists a polynomial-time algorithm for *at least one* non-polynomial problem, *all* NP problems are automatically "P"
  - Prove "P=NP": Insta-Nobel Prize. Guaranteed!

In fact, NP is "useful"

- Public-key encryption (e.g., SSL/ssh) largely works on the fact that decrypting an encrypted message takes an extraordinarily long time
  - Details beyond scope of class
- If someone were to prove that P=NP, many of today's encryption algorithms would have to be thrown out the window
- Fortunately, no one has come close to proving it
- But no one has come close to proving the opposite either

So where do we go from here?

- Most computer scientists (except great theoreticians) focus on making new computable algorithms, hopefully in polynomial time
- With the knowledge you've learned in this class, you have the pieces to go ahead and build such algorithms, and code them
- Remaining CS classes introduce advanced concepts, but they still boil down to the same thing

Next time

- No next time ☹
- In labs:
  - C – Modularity, Makefiles
  - Java – packaging, Java API
- Final two weeks from today
- Wait! We're not finished

Final

- Structure: very similar to midterm, about 50% longer – so you shouldn't need all three hours
- I *will* put up a reading list by the end of this week that will cover section reading in great detail
  - Will tweak slides to remove stuff we didn't get to in class…
- Review sessions next week: they'll be open-ended, so bring questions!

Feedback

- This class, as I said at the beginning, is experimental
- *Please* fill out the SEAS Oracle survey
  - http://oracle.seas.columbia.edu
  - You can win an iPod!
- But let's also discuss the class now
  - I'm writing a report, and what you tell me can help
  - Final exam bonus anonymous survey?

Thank you!

- You guys have been a great audience.
- I hope you found this class rewarding.
- Good luck with the rest of your Computer Science mini-careers!
  - And with finals
- Don't forget review sessions next week