Computability and Computational Complexity

Lecture 23

P & NP

Computation Problems

- A discrete computational task can be modelled as that task of evaluating a function $f : \mathbb{N} \to \mathbb{N}$, or $f : \{0,1\}^* \to \{0,1\}^*$

  Set of all finite length binary strings
- Decision problems: output is a single bit ("yes" or "no")
  - $f : \{0,1\}^* \to \{0,1\}$. $L_f \triangleq \{ x \mid f(x) = 1 \}$ is called the language associated with the decision problem f
- More complex notions exist
  - Interactive (or reactive) computation: inputs can be fed after outputs are observed
  - Multiparty computation: inputs and outputs are distributed among many automata that interact with each other
- We will focus on decision problems

Uniform & Non-Uniform Computation

- What is a program for computing a function?
- Uniform: A finite length string that encodes an automaton (in a "standard" model of computation)
  - Same program can be fed inputs of any length
- Non-Uniform: A different program allowed for each input length
  - The full program is an infinite string encoding { P_0, P_1, P_2, ... }
  - Unrealistic model
  - A function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be represented by a bit-string of length 2^n (truth table). P_n can simply have this string hardcoded into it
  - Interesting question for non-uniform computation: How fast <u>and</u> small can the program P_n be for a given function f?

Uncomputability

- A decision problem, $f : \{0,1\}^* \longrightarrow \{0,1\}$: An infinite string, encoding L_f
- A (uniform) program: a finite string
- There are only countably many programs, but there are uncountably many problems!
  - For most problems, there is no program computing it!
- This argument works irrespective of the details of the model of computation
  - Q1: Does the choice of the model affect which functions are computable and which are not?
  - Q2: Most of the uncountably many uncomputable problems are "uninteresting." Are there interesting problems that are uncomputable?

Uncomputability

- Does the choice of the model affect which functions are computable and which are not?
- Not really!
- Several standard models of computation have been proposed, but they are powerful enough to simulate each other
  - Examples: Lambda Calculus, Turing Machines, and Random Access Machines
- Church-Turing thesis: The standard models so far (which are all equivalent to each other) are the only models of "effective" (physically realisable) computation

The Uncomputable

- Are there interesting problems that are uncomputable?
  - Yes!
- Hilbert's 10th problem: find an algorithm to check if a "Diophantine equation" has a solution

shown uncomputable in 1970

  - i.e., check if there is an integer solution to all the variables in a polynomial. (e.g., the ones in Fermat's last theorem, $x^3+y^3=z^3$, $x^4+y^4=z^4$, ...)
- Hilbert's Entscheidungsproblem: given a statement in first order logic, check if it is true/provable

shown uncomputable by Church and Turing [1936]

  - (In first order logic, true iff provable)
- The Halting Problem: Given a (program,input) pair decide if the program halts or not

Turing [1936]

- ...

Computational Complexity

- Computability theory deals with what can be computed (in various models of computation)
- Computational Complexity Theory deals with the amount/nature of resources needed for solving computable problems
- <u>Time Complexity of a problem</u>: minimum running time needed by <u>any</u> program to solve n-bit instances of a problem (in the worst-case: i.e., max over all instances)

Computational Complexity

- <u>Time Complexity of a problem</u>: minimum running time needed by any program to solve n-bit instances of a problem (worst-case: max over all instances)
- Some computational problems take a long time to solve, simply because the solutions are long
  - e.g., Tower of Hanoi (exponentially many moves)
- But some problems can be hard, even if the output is short — say, a single bit!
  - Recall: Such problems (decision problems) can even be uncomputable!
  - We will focus on computational complexity of decision problems

Computational Complexity

- Church-Turing thesis: Computability of a problem doesn't depend on the exact choice of the model (as long as it is as powerful as a Turing machine)
- How about computational complexity of a problem?
- Model does matter (a bit)
- But mostly, polynomial-time computation in one model is polynomial-time computable in another model (with a different polynomial)
  - But (probabilistic) Turing Machines are not known (or believed) to be able to simulate computation in a "Quantum Turing Machine" with polynomial overhead
- But we will stick to non-quantum models

Polynomial Time

- P: class of decision problems which have polynomial time algorithms
  - Extended Church-Turing thesis: if polynomial time in any "effective" (realizable) and deterministic computational model, then polynomial time in the Turing Machine model
- What we really care about is having fast algorithms: typically $O(n^2)$, $O(n \log n)$, $O(n)$, sub-linear etc.
- But since the exact polynomial depends on the computational model (e.g., random access memory vs. sequential), P is used as a robust notion that doesn't change with the model
  - If complexity is polynomial (i.e., $O(n^c)$) in a (non-quantum) model, then remains polynomial in all (reasonable) models

NP

- Class of decision problems which have polynomial time algorithms <u>when given some help</u>
- NP : non-deterministic polynomial time
  - $P \subseteq NP$ (need not use the help)
- What kind of help? Guidance on what "paths" to explore during computation
  - Non-deterministic: multiple ways in which computation can proceed at each step

P & NP: an analogy

- Solving a computational problem is like a treasure-hunt
  - When you follow an algorithm, you are moving through an infinite state-space, starting from a state defined by the problem instance, until you hit the solution, if it exists (or find out that no solution exists)
  - Polynomial time algorithm: no matter what the input is, if a solution exists, it reaches one in $O(n^c)$ steps
  - Non-deterministic polynomial time algorithm: if a solution exists, if someone could <u>guide the algorithm at every turn</u>, it will reach a solution in $O(n^c)$ steps (or realize that it was misguided)
    - i.e., if a solution exists, a <u>short & verifiable</u> path to a solution exists. (Needn't be easy to find it without guidance.)

P & NP: an analogy

- E.g., checking if a (connected) graph is 2-colorable
- Nodes are coloured one-by-one, until all coloured, or a contradiction found
- No such algorithm known for 3-colourability!

```
2colourable (G: connected graph) {
   Q := empty-list
   s := an arbitrary node in G
   colour[s] := 0; insert(Q, s)
   while (Q not empty) {
      x := pop(Q)
      c := colour[x]
      for each neighbour y of x
         if (colour[y] = c)
            return false
         if (y uncoloured)
            colour[y]:=1-c; insert(Q, y)
   }
   return true
}
```

- But if G is 3-colourable, there exists a short & efficiently verifiable path to valid colouring (colour first and verify edges one-by-one)
  - Guidance: which colour to use for each node

1

ZUPF

Question

- Let 2COL and 3COL stand for the decision problems of 2-colourability and 3-colourability of graphs. Consider the statements:

  (1) 2COL $\in$ P (2) 2COL $\in$ NP

  (3) 3COL $\in$ P (4) 3COL $\in$ NP

  Then which statements do we know to be true?

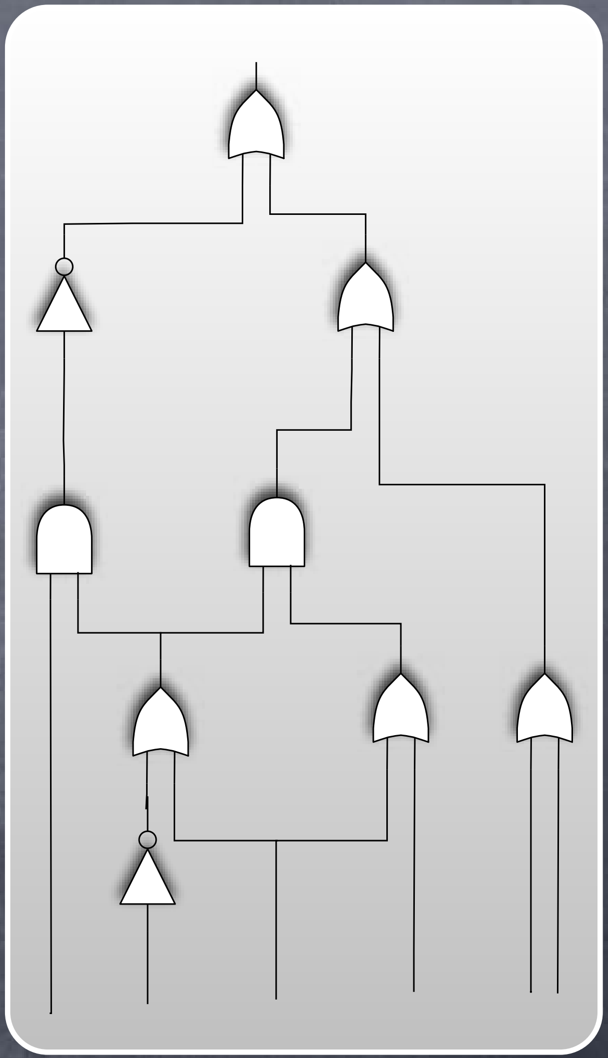
A. All statements
B. Only (1) and (4)
C. Only (1), (2) and (4)
D. Only (2) and (3)
E. Only (1) and (2)

NP: Alternate View

- An alternate equivalent definition of NP: without the notion of guidance
- There is a polynomial-time algorithm to verify a "certificate" that a solution exists (if it exists)
  - E.g., certificate is the 3-coloring of a graph. Verifier checks that every edge is satisfied with the coloring
  - Decision problem ≡ ∃ cert s.t. Verify(instance,cert) ?
  - Note: there may not be a certificate to prove (to a polynomial time verifier) that no solution exists
    - **co-NP**: Class of problems with poly-time verifiable counter-examples (certificate of "no" being the answer)
  - e.g., 3COL ∈ NP, but not known to be in co-NP

Example: Boolean Circuits

- A directed acyclic graph: Boolean valued wires, AND, OR, NOT gates, inputs, output
  - Circuit evaluation CKT-VAL: given circuit C and inputs x, find C(x) (i.e., C's boolean output value, on input x)
  - Can be done very efficiently: if done in the right order, evaluating each wire takes O(1) time. CKT-VAL is in P.
- CKT-SAT: given circuit C, is there a "satisfying" input for C (s.t. output=1)? i.e., $\exists x\ C(x)=1$? In NP.
- $\overline{\text{CKT-SAT}}$: given C, is it that there is no satisfying input. i.e., $\forall x\ C(x)=0$? In co-NP.



P vs. NP

- The Million Dollar Question: is P=NP?
  - We know P⊆NP, so the question is if every problem in NP is in P
    - Or are there problems where guidance really helps?
  - Generally believed: P≠NP
    - In particular, graph 3-colourability and CKT-SAT believed not to have polynomial time algorithms
- Also open is NP = co-NP?

NP-completeness

- Graph 3-colourability, CKT-SAT and several other problems in NP are tightly related to each other
  - If any one of them is in P, then all of them are in P !
  - Further, then P = NP !
- Proving P≠NP is equivalent to proving (say) CKT-SAT $\notin$ P
  - And proving P=NP is the same as proving CKT-SAT $\in$ P
- NP-Complete problem: Any problem in NP can be **reduced** to it in polynomial time
  - Reducing Problem 1 to Problem 0: Given an instance X of Problem 1, convert it to an instance Y of Problem 0, s.t. X has answer yes iff Y has answer yes

NP-completeness

- Proving P=NP is the same as proving CKT-SAT $\in$ P
  - About 50 years (and counting) of failed attempts at finding polynomial-time algorithms for <u>any</u> of the NP complete problems
- Several practically important problems are known to be in NP or co-NP, but not known to be in P.
  - Related to finding the smallest circuitry for a device, finding optimal airline scheduling, breaking encryption schemes, ...
- Widely believed that P≠NP, but no techniques to prove that

Zoo of Complexity Classes