CS125 **Lecture 13** **Fall 2016**

13.1 Nondeterministic Polynomial Time

Now we turn to incorporating nondeterminism in universal models of computation, namely Turing Machines and Word-RAMs.

For Nondeterministic Turing machines, we do it just like for NFAs, now allowing multiple transitions on each state-symbol pair. That is, we allow the transition function to be a mapping $\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$.

The generalization of computation is also analogous to what we did for NFAs.

- For a configuration $C = uq\sigma v$ (where $u, v \in \Gamma^*, q \in Q, \sigma \in \Gamma$), we write $C \Rightarrow_M C'$ for every configuration C' that can be obtained by applying one of the transitions in the set $\delta(q, \sigma)$ to C. (So C' is not uniquely determined by C.)
- An NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *accepts* $w \in \Sigma^*$ if there *exists* a sequence $C_0, \ldots, C_t$ of configurations such that
  - $C_0 = q_0 w$,
  - $C_{i-1} \Rightarrow_M C_i$ for each $i = 1, \ldots, t$, and
  - In C_t, M is in state q_{halt} and the contents of the tape to the left of the first blank symbol is the symbol 1.
- For an NTM M, the *language recognized by* M is $L(M) = \{w : M \text{ accepts } w\}$. That is, for every $w \in L(M)$, there is at least one accepting computation path of M. And for every $w \notin L(M)$, all computation paths either halt in a non-accepting configuration or run forever.
- An NTM M *decides* a language L if $L(M) = L$ and M has no infinite computation paths. (So M halts on every input and every computation path, without loss of generality always having an output of 1 or 0.)

Note that the only types of computational problem we consider for nondeterministic algorithms are *decision problems* (i.e. languages).

One way to incorporate nondeterminism into the Word-RAM model is via the GOTO command:

- We allow instructions of the form "IF $R[i] = 0$ GOTO ℓ_1, ℓ_2, ℓ_3, …, or ℓ_k." (Each choice yields a distinct computation path.)

- For a configuration $C = (\ell, S, w, R, M)$ of a word-RAM program P in which P_ℓ is such a GOTO command, there may be k different configurations $C' = (\ell', S, w, R, M)$ such that $C \Rightarrow_P C'$, namely ones for each $\ell' \in \{\ell_1, \ldots, \ell_k\}$.

For the next few lectures, we will be focused on the extent to which nondeterminism increases the power of polynomial-time algorithms:

Definition 13.1 *The running time $T(n)$ of a nondeterministic algorithm A (whether a TM or a Word-RAM) is the maximum number of steps to reach a halting configuration over all inputs of length n and all computation paths.*

NTIME$(T(n))$ *is the class of languages decided by algorithms running in time $O(T(n))$, and* nondeterministic polynomial time *is the class*

$$\mathrm{NP} = \bigcup_c \mathrm{NTIME}(n^c).$$

It can be verified that the polynomial equivalence we proved between TMs and Word-RAMs extends to their nondeterministic analogues, so the class NP does not depend on which model we use, but the finer classes NTIME(n^c) might depend on the model.

An Example: Travelling Salesman Problem

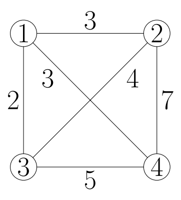
Clearly P $\subseteq$ NP. But there are problems in NP that are <u>not obviously</u> in P ($\neq$ "obviously not"). Such as the TRAVELLING SALESMAN PROBLEM:

- Let $m > 0$ be the number of <u>cities</u>,
- Let D be an $m \times m$ matrix of nonnegative real numbers giving the <u>distance</u> $D(i, j)$ between city i and city j, and
- let B be a distance <u>bound</u>

Then

$$\mathrm{TSP} = \{\langle m, D, B\rangle : \exists \text{ tour of all cities of length} \leq B\}.$$

To illustrate:



$n = 4$

$B \geq 15 \Rightarrow \langle m, D, B \rangle \in \text{TSP}$
$B \leq 14 \Rightarrow \langle m, D, B \rangle \notin \text{TSP}$

"tour" = visits every city and returns to starting point

There are many variants of TSP, eg require visiting every city exactly once, triangle inequality on distances...

Proposition 13.2 TSP $\in$ NP

Proof: If $\langle m, D, B \rangle \in$ TSP, the following nondeterministic algorithm will accept in time $O(n^3)$, where $n =$ length of representation of $\langle m, D, B \rangle$.

- nondeterministically write down a sequence of cities $c_1, \dots, c_t$, for $t \leq m^2$. ("guess")
- trace through that tour and verify that all cities are visited and the length is $\leq B$. If so, halt in q_{accept}. If not, halt in q_{reject}. (and "check")

Conversely, if $\langle m, D, B \rangle \notin$ TSP, above has no accepting computations. ■

But we do not know if TSP $\in$ P. Indeed, all known deterministic algorithms for TSP take exponential time (in the worst case).

A useful characterization of NP

Def: A <u>verifier</u> for a language L is a (deterministic) algorithm V such that $L = \{x : V \text{ accepts } \langle x, y\rangle \text{ for some string } y\}$.

Def: A <u>polynomial-time</u> verifier is one that runs in time polynomial in $|x|$ on input $\langle x, y\rangle$.

A string y that makes $V(\langle x, y\rangle)$ accept is a "proof" or "certificate" that $x \in L$.

Example: TSP

certificate y = the sequence $c_1, \ldots, c_t$ of cities from the proof of Proposition 13.2.

$V(\langle x, y\rangle)$ will check that this sequence indeed gives a tour, and the total cost is at most B.

N.B. Without loss of generality, $|y|$ is at most polynomial in $|x|$.

Theorem 13.3 NP *equals the class of languages with polynomial-time verifiers.*

Proof:

$\Leftarrow$

Suppose L has a poly-time verifier V which runs in time at most $T(n) \leq Cn^k$ for some constant $C, k > 0$. Then our NTM M will nondeterministically guess a witness y of length at most Cn^k, halt with the output of $V(x, y)$ (that is, accepting iff $V(x, y) = 1$).

$\Rightarrow$

Suppose L is in NP. Then there is an NTM M deciding L. Over the course of running M on some input x of length n, M runs for at most Cn^k steps. Our verifier V then works as follows. The witness y will be a sequence of at most Cn^k triples (q, σ, D) where $D \in \{L, R\}$, corresponding to transitions of the Turing Machine. The jth triple tells us which element of the output of the transition function δ we should take in the jth step when running M on x, i.e. it specifies one computation path for M running on x. The verifier V verifies that y represents a valid sequence of transitions as per the description of M and the input x, and returns 1 iff this computation path leads to M accepting x.

■

"L is in NP iff members of L have short, efficiently verifiable certificates"

More problems in NP

- $L_{\text{fact}} = \{\langle N, M\rangle : N \text{ has a factor in } \{2,3,\ldots,M\}\}$.
  - As an exercise, try to show that that this language is in P iff there is a polynomial-time algorithm for FACTORING. It is conjectured that no such algorithm exists (and indeed, much cryptography in use relies on this conjecture.) Recall that here we are talking about time polynomial in the *length* of the input, i.e. time $\text{poly}(\log N)$.
  - However, it is easy to see that $L_{\text{fact}} \in \text{NP}$: the witness is simply the factor.

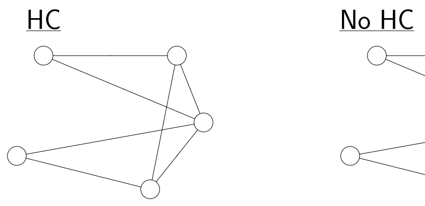
- For contrast, the very related language NONPRIMES $= \{\langle N\rangle : N \text{ has a factor in } \{2,\ldots,N-1\}\}$ is known to be in P! This is not an an easy result; it was only shown fairly recently in the 2000's [AgrawalKS04].
  - It is easier to see the following. First note NONPRIMES is in NP since a witness is simply a nontrivial factor of N. This means that the complement language PRIMES $= \{\langle N\rangle : N \text{ is a prime integer}\}$ is in a complexity class called *co-NP*. co-NP is the class of languages L for which there is a poly-time verifier V such that $L = \{x : \forall y,\ V(x,y) = 0\}$ (V runs in time polynomial in $|x|$). That is, there is no witness convincing V that x is in the language.
  - An easier thing to showing NONPRIMES $\in P$ is to show that PRIMES $\in$ NPSo in fact it is in NP $\cap$ co-NP! The fact that it is in NP was first shown by Vaughan Pratt in 1975.
  - The idea to show that it is in NP is to use Lehmer's theorem (which we won't prove here — it uses algebra, so if you want to know more then take MATH 122). Lehmer's theorem states that N is prime iff there exists an integer $1 < a < N$ such that (1) $a^{N-1} \equiv 1 \pmod N$, and (2) for all prime divisors q of $N-1$, $a^{(N-1)/q}$ is *not* equivalent to 1 mod N.
  - If you take Lehmer's theorem as a black box, then the Pratt certificate simply lists a together with all the prime divisors $p_1,\ldots,p_k$ of $N-1$. It then *recursively* lists such a certificate for each of these divisors $p_1,\ldots,p_k$ (to convince the verifier that these p_i are in fact prime), so that there is in fact a certificate "tree". The base case of the recursion, or leaves of the tree, correspond to $p_i = 2$. One can show by induction on $n \geq 3$ that the number of non-leaf nodes in the tree rooted at n is at most $4\log n - 4$ for

$n \geq 3$. Note also that each p_i and a take at most $O(\log N)$ bits to write down, and thus the total size of the certificate is $O(\log^2 N)$, which is polynomial in the binary encoding of N, which takes $O(\log N)$ bits.

- The verifier checks (1) and (2) given the certificate, and for each prime divisor q of N, the verifier recursively checks that q is prime given the information in the certificate. Note that $a^k \bmod N$ can be computed in $O(\log k)$ multiplications by repeated squaring. In particular if we write k in binary as $k = \sum_{j \in S} 2^j$, then $a^k \bmod n = a^{\sum_{j \in S} 2^j} \bmod n = \prod_{j \in S} a^{2^j} \bmod n$. We can form all these powers a^{2^j} modulo n by repeated squaring, and there are at most $\log_2 k$ of them of interest.

- HAMILTONIAN CIRCUIT

$$\text{HC} = \{G : G \text{ an undirected graph with a circuit that touches each node exactly once}\}.$$



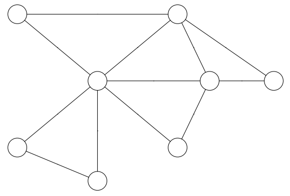
Really just a special case of TSP. (why?) (Recall that we are not fussy about the precise method of representing a graph as a string, because all reasonable methods are within a polynomial of each other in length.)

For nearly 50 years, the best known algorithm for Hamiltonian Cycle detection was the $O(n^2 2^n)$ dynamic programming algorithm we saw earlier in the course, due independently to Bellman, and to Held and Karp, both in 1962. An improvement was only made very recently by Björklund [Bjorklund10] $O(n^k 1.657^n)$ (which is of course $O(1.658^n)$). In the case the graph is bipartite, he gave a further improved bound to $O(n^k 1.414^n)$. For the TSP problem though, nothing is known better than $O(n^k 2^n)$. An open research problem: can TSP be solved in time $O(1.999999^n)$?

All known algorithms for HC take exponential time, so we might conjecture that HC $\notin$ P. But consider the very similar problem EULERIAN CIRCUIT:

$$\text{EC} = \{G : G \text{ is an undirected graph with a circuit that passes through each } \underline{\text{edge}} \text{ exactly once}\}$$

.



It turns out that G is Eulerian iff G is connected and every vertex has even degree! (Proven in AM107.) So EC $\in$ P.

- SATISFIABILITY

  Def: A Boolean formula (B.F.) is recursively defined as any of the following:

  · a "Boolean variable" $x, y, z, \ldots$

  · $(\alpha \vee \beta)$ where α, β are B.F.'s.

  · $(\alpha \wedge \beta)$ where α, β are B.F.'s.

  · $\neg\alpha$ where α is a B.F.

  e.g. $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$

  [Omitting redundant parentheses]

  Given a boolean formula φ and a truth assignment a : Boolean variables $\to \{0,1\}$, the evaluation $\varphi(a)$ is defined in the natural way using the rules of boolean logic.

  Prop: SAT $= \{\varphi : \varphi$ a B.F. such that $\exists a \; \varphi(a) = 1\}$ is in NP.

13.2 The P vs. NP Problem

- We would like to solve problems in NP efficiently.
- We know P $\subseteq$ NP.
- Problems in P can be solved "fairly" quickly.
- What is the relationship between P and NP?

NP and Exponential Time

Claim: NP $\subseteq \bigcup_k \text{TIME}(2^{n^k})$

Proof: If $L \in$ NP, then $L \in \text{NTIME}(n^k)$ for some k. If NDTM M decides L in time n^k, then the standard simulation of M by a DTM runs in time $O(c^{n^{k+1}})$.

(Check the c^{n^k} computations, each of length n^k).

Of course, this gets us nowhere near P. Does P = NP? That is, do all the NP problems have polynomial time algorithms? It doesn't "feel" that way but as of today there is no NP problem that has been proven to require exponential time!

The Strange, Strange World if P = NP

Thousands of important languages can be decided in polynomial time, e.g.

- SATISFIABILITY
- TRAVELLING SALESMAN
- HAMILTONIAN CIRCUIT
- MAP COLORING
- $\vdots$

If P = NP**, then Searching becomes easy**

Every "reasonable" search problem could be solved in polynomial time.

- "reasonable" $\equiv$ solutions can be recognized in polynomial time (and are of polynomial length)
- SAT SEARCH: Given a satisfiable boolean formula, find a satisfying assignment.
- FACTORING: Given a natural number (in binary), find its prime factorization.
- NASH EQUILIBRIUM: Given a two-player "game", find a Nash equilibrium.
- $\vdots$

If P = NP**, Optimization becomes easy**

Every "reasonable" optimization problem can be solved in polynomial time.

- Optimization problem $\equiv$ "maximize (or minimize) $f(x)$ subject to certain constraints on x" (AM 121)
- "Reasonable" $\equiv$ "f and constraints are poly-time"
- MIN-TSP: Given a TSP instance, find the shortest tour.
- SCHEDULING: Given a list of assembly-line tasks and dependencies, find the maximum-throughput scheduling.

- PROTEIN FOLDING: Given a protein, find the minimum-energy folding.
- INTEGER LINEAR PROGRAMMING: Like linear programming, but restrict to integer
- CIRCUIT MINIMIZATION: Given a digital circuit, find the smallest equivalent circuit. (Is "reasonable" if P = NP.)

If P = NP**, Secure Cryptography becomes impossible**

Every polynomial-time encryption algorithm can be "broken" in polynomial time.

- "Given an encryption z, find the corresponding decryption key K and message m" is an NP search problem.
- Thus modern cryptography seeks to design encryption algorithms that cannot be broken under the *assumption* that certain NP problems are hard to solve (e.g. FACTORING).
- Take CS 127.

If P = NP**, Artificial Intelligence becomes easy**

Machine learning is an NP search problem

- Given many examples of some concept (e.g. pairs (image1, "dog"), (image2, "person"), ...), classify new examples correctly.
- Turns out to be equivalent to finding a short "classification rule" consistent with examples.
- Take CS228.

If P = NP**, even Mathematics becomes easy!**

Mathematical proofs can always be found in polynomial time (in their length).

- SHORT PROOF: Given a mathematical statement S and a number n (in unary), decide if S has a proof of length at most n (and, if so, find one).
- An NP problem!
- cf. letter from Gödel to von Neumann, 1956.

Gödel's Letter to Von Neumann, 1956

[$\phi(n) =$ time required for a TM to determine whether a mathematical statement has a proof of length n]

…

If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$) this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. …

It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search. …

References

[1] Manindra Agrawal, Neeraj Kayal, Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2), pages 781–793, 2004.

[2] Andreas Björklund. Determinant Sums for Undirected Hamiltonicity. *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 173–182, 2010.