CSCI 341–Fall 2024: Lecture Notes
Set 18: P & NP

Edward Talmage

December 2, 2024

Definition 1. An algorithm is *polynomial time* if its time complexity $t(n)$ is $O(n^k)$ for some constant k. The *class* P is the set of all languages which can be decided in polynomial time.

$$P = \{L \subseteq \Sigma^* \mid \text{there exists a TM deciding } L \text{ in polynomial time}\}$$

Alternately, $P = \bigcup_{k \in \mathbb{R}^+} Time(n^k)$.

> **Aside:** Recall that $f(n) = O(g(n))$ iff there exists some $n_0, c > 0$ s.t. $\forall n \geq n_0, f(n) \leq cg(n)$. That is, eventually $f(n)$ is less than a constant factor times $g(n)$ forever.

Definition 2. A *polynomial-time reduction* R is a reduction with time complexity $O(n^k)$, for some constant k. We say that $L \leq_p M$ if there is a poly-time reduction from L to M.

Theorem 1. *If $L \leq_p M$ and $M \in P$, then $L \in P$.*

Proof. By construction. Let R be a reduction from L to M that runs in $O(n^k)$ time and A be an algorithm deciding M in $O(n^h)$ time. Construction the following decider for L:

On input w:

1. return $A(R(w))$

If $|w| = n$, then $|R(w)| = O(n^k)$, since R takes only $O(n^k)$ steps, so cannot leave an output longer than that. It then follows that $Time(A(R(w))) = Time(A(O(n^k))) = O((n^k)^h)) = O(n^{kh})$, which is polynomial in n, so $L \in P$. □

We care about the class P because all "reasonable", deterministic models of computation we have invented are polynomially reducible to TMs. That is, they have polynomial time solutions to a problem iff there is a poly-time TM for that problem. For example, we did this with converting k-tape TMs to standard TMs. Our runtime squared, but that is only a polynomial change. For this reason, we generally consider polynomial-time algorithms to be efficient, and do so independent of computational model. This is a very high-level view of efficiency, and we would usually care about the difference between n^2 and n^{10000}, but even n^{10000} is asymptotically faster than 2^n, so this is still an interesting division to consider.

Example:

$$PATH = \{\langle G, s, t\rangle \mid G = (V, E) \text{ is a graph, } s, t \in V, \exists \text{ path } s \rightsquigarrow t \text{ in } G\}$$

This is a the decision problem asking whether there is a path in a given graph between two given nodes.

> **Exercise:**
>
> 1. What is the input size?
> 2. Can you give a simple but inefficient algorithm for determining whether there is a path from s to t?
> 3. Can you give an efficient algorithm for $PATH$? Give an implementation-level description.
> 4. What is your algorithm's time complexity?

Input size depends on graph representation, but they are all polynomially equivalent (we could convert between them in time polynomial in the length of the representation). A simple algorithm would be to list all possible paths by permuting all vertices (we know cycles are not necessary, since we are just considering reachability), then check each to see if it contains a subpath from s to t. But this takes something like $\Omega(|V|!)$ time, so we need a better algorithm.

Recall Breadth-First Search. Starting from s, mark all neighbors, then explore outward, marking each reached node. If we mark t, accept. If we never mark t before being unable to explore any new vertices, reject. We can also give a TM description:

On input $\langle G, s, t\rangle$, where G is a graph, s, t vertices in G:

1. Mark s
2. Until you mark no additional nodes:
   (a) Look at each edge (a, b) of G. If a is marked and b is not marked, mark b.
   (b) If marked t, accept
3. Reject

Let n be the length of the input, assume the graph is represented simply as a list of vertices, then a list of edges. It takes $O(n^2)$ time to mark s, since we have to walk down the tape past G to see what vertex is the starting point, then walk back to mark it, possibly zig-zagging if vertex descriptions are relatively long. Step 2 will loop at most n times, since we terminate if we mark no new vertices. The loop in Step 2.a will run at most $O(n)$ times, since there are at most n edges. Each iteration would take $O(n^2)$ to find the next edge, check that a is marked, and go mark b. The rest of the algorithm takes constant time, so we have a total of $O(n^4)$. This is probably not a tight bound, but it tells us that the problem is solvable in polynomial time, which is all we care about right now.

> **Aside:** A caution on input size and encodings: If the input is an integer n (or really any numerical value) and the algorithm take n steps, it is exponential, not polynomial. To pass n as an input, we must encode it in some base b. This requires $\Theta(\log_b n)$ tape cells, so a time complexity of n is $O(b^{\log_b n}) = O(b^\ell)$, where ℓ is the input length. Such time complexities are known as *pseudo-polynomial*, and are not as efficient as our intuition tends to feel they should be.

Example:

$$HAM = \{\langle G, s, t\rangle \mid G \text{ is a graph, } s, t \text{ vertices in } G, \text{ and } G \text{ has a Hamiltonian path } s \rightsquigarrow t\}$$

A *Hamiltonian path* is one which visits each node in the graph exactly once. We can solve this problem expensively by extending our inefficient solution to $PATH$ such that when checking all possible paths, we only accept if a path ends and t and is Hamiltonian. This would take something like $O((|V|-2)!)$ time, since each potentially Hamiltonian path is a permutation of the vertices which starts with s and ends with t. We do not currently know whether there is an efficient algorithm for HAM. More generally, there is no known proof that HAM is in or not in P.

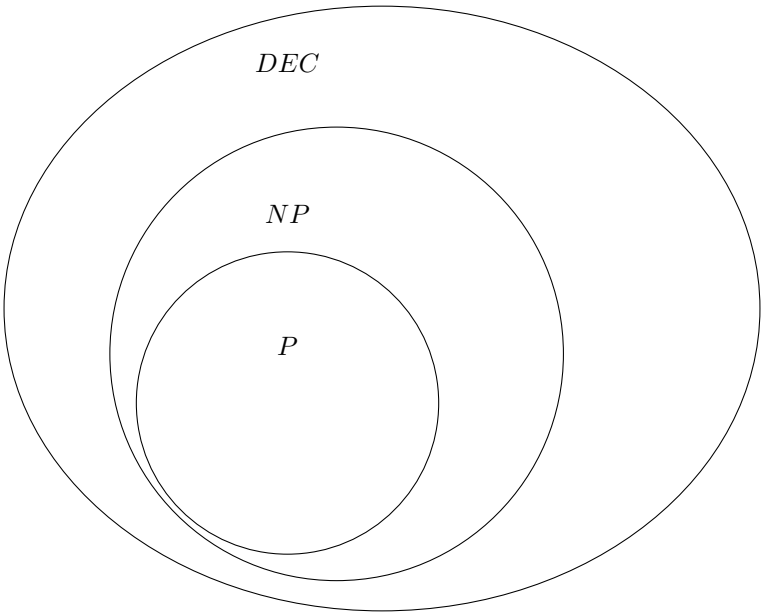
1 Non-Determinism

Using non-deterministic Turing machines, though, we can fairly straightforwardly solve HAM in polynomial time, since we can non-deterministically choose whether the next node in the input is the next node in the potential path, then if any such permutation is a valid Hamiltonian path, we accept.

Definition 3. An algorithm is *non-deterministic polynomial time* if its time complexity $t(n)$ on a non-deterministic TM is $O(n^k)$ for some constant k. The *class* NP is the set of all languages which can be decided in non-deterministic polynomial time.

$$NP = \{L \subseteq \Sigma^* \mid \text{there exists an NTM deciding } L \text{ in polynomial time}\}$$

This gives us a new view of the world of decidable languages:



We can see that $P \subseteq NP$, since every deterministic TM is a non-deterministic one which simply has only a single choice at each step. The question of strict containment, however, specifically whether there is an element in NP which is not also in P, is probably the most famous open problem in computer science. In fact, it is one of the Clay Millennium problems (see `https://www.claymath.org/millennium-problems`), identified as significant problems in mathematics and for which there is a $1 million prize if anyone is able to resolve it. The general thought is that $P \neq NP$, meaning that there are problems solvable in polynomial time on non-deterministic machines which cannot be solved in polynomial time on deterministic machines. The idea for this intuition is that a proof that $P = NP$ could be much easier, since a single polynomial-time algorithm for a known-hard problem could collapse NP to equal P, while a proof that $P \neq NP$ involves proving that no technique, however clever, could ever produce a deterministic polynomial solution. Since no one has (yet) found a polynomial solution for one of these problems, we suspect that it may be impossible.

Exercise: How long does it take to check a solution to HAM? That is, if I give you a graph and a claimed Hamiltonian path, how long would it take you to check that my claimed solution is correct?

We simply need to walk along the given path, verifying that it is a valid path, with edges connected one after another, that all edges are present in G, and that it starts with s and ends with t. This is linear complexity, and deterministic. In general, this idea will give us an alternate definition for NP, which is useful.

Definition 4. A *verifier* for a language A is a decider V such that

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

This definition seem backwards, since it looks like a definition for A, but is actually a definition for V. What we are saying is that a string w is in A if and only if we can pair it with some other string c such that V accepts the pair v, c. We call c a *certificate* and it usually think of them as sample solutions, proving by construction that w is a string which is in A. We can see this with the Hamiltonian path problem, where w is the graph description and c is the claimed Hamiltonian path. We can then check that c is a valid path to verify that w describes a graph with a Hamiltonian path. Formally, that verifier is the TM

On input $\langle\langle G, s, t\rangle, c\rangle$:

1. Ensure the first edge in c is (s, v) for some $v \in G$ and (s, v) is an edge in G. Mark s and v.
2. Walk through all remaining edges $(v, w) \in c$:
   (a) If (v, w) is not in G, reject.
   (b) If v is not the end of the previous edge in c, reject.
   (c) If w is marked, reject.

(d) Else, mark w.

3. If the last edge in c does not end at t, reject.

4. If there are any unmarked vertices in G, reject. Else, accept.

We can use verifiers to give an alternate definition of NP as the class of languages which are verifiable in polynomial time. This is, perhaps, the more popular definition, since it sticks to the deterministic model with which computer scientists tend to be more familiar. The name comes from non-determinism, though, as NP stands for Non-deterministic Polynomial time.

Claim 1. *A language is in NP iff it is polynomially verifiable (there is a polynomial-time verifier for it).*

Proof. ($\Leftarrow$) First, consider the reverse implication: Assume that we have a poly-time verifier V for language A. Let k be such that V decides in n^k time. We construct an NTM for A:

On input w of length n:

1. Nondeterministically select string c of length at most n^k.

2. Run V on input $\langle w, c\rangle$.

3. If V accepts, accept, else reject.

This will accept iff there is a string c^* that verifies w, since there will be some non-deterministic branch which runs $V(\langle w, c^*\rangle)$, which will accept. Step 1 takes n^k time to generate all possible certificates, Step 2 takes n^k time, since that is an upper bound on the runtime of V. Step 3 is constant time, so we have a total of $O(n^k)$. Thus, we have an NTM deciding A in polynomial time.

($\Rightarrow$) Second, assume N is a poly-time NTM deciding A. We will construct a polynomial-time verifier V:

On input $\langle w, c\rangle$, where w and c are strings:

1. Simulate $N(w)$, using each character of c to choose the non-deterministic branch to follow. (Similar to the address tape in our simulation of NTMs.)

2. If the explored branch of N's computation accepts, accept. Else reject.

This runs for $O(|c|)$ iterations, since we use each symbol of c to control one step of N. Each iteration is polynomial time, since it is a single step of N, though we may have to walk across the tape to choose the correct computation branch, so the total time is polynomial.

If there is a way for N to accept w, then there must be a c that makes this verifier accept, since it is a description of that (finite) path through N. This certificate would probably not be human-readable, since it is tightly tied to the inner workings of N, but it must exist and we only care that there is *some* certificate for each string $w \in A$. □

Example: $SUBSET\_SUM \in NP$, where $SUBSET\_SUM$ is the problem of determining whether a given set has a subset which sums to given target value:

$$SUBSET\_SUM = \left\{ \langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \exists \{y_1, \ldots, y_h\} \subseteq S s.t. \sum_i y_i = t \right\}$$

Exercise: Give a verifier and a polynomial-time non-deterministic machine for the language. (Half of class do each.)

One possible verifier for $SUBSET\_SUM$ is

On input $\langle \langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of elements of S

2. Test whether c's sum is t

3. If both succeed, accept. Else, reject

One possible poly-time NTM for $SUBSET\_SUM$ is

On input $\langle S, t \rangle$:

1. Non-deterministically select a subset c of the values in S
2. Test whether c's sum is t
3. If so, accept. Else, reject.