Introduction to Computational Complexity

February 7, 2024

Given a graph G and two vertices $x, y \in V(G)$, is there a path from x to y? This is an example of a computational problem. These problems have variable input (in this case the graph G) and an output. If the output is either yes or no (as is the case here) then the problem is called a *decision problem*.

Write $\{0,1\}^*$ for the set of 0-1 strings of arbitrarily (finite) length, i.e. the set $\bigcup_{n=1}^{\infty} \{0,1\}^n$. Then a decision problem can, in principle be encoded as a Boolean function, that is, a function $f\colon \{0,1\}^* \to \{0,1\}$. In our problem we could, for example, encode the graph G by its adjacency matrix, which will give a string of 0s and 1s representing the input.

The set $\{x \in \{0,1\}^* \mid f(x) = 1\}$ is called the *language* defined by f (n.b. this is a silly name).

Turing Machines

A k-tape Turing machine consists of the following items and rules.

- A finite set A called the *alphabet*.
- A collection of k *tapes*, where a tape is an infinite sequence (indexed by $\mathbb{N}$) of *cells*, and each cell contains an element of A.
- A finite set S of *states*, containing two special states, namely S_{init} and S_{halt}. A state is a function $A^k \times S \to A^k \times S \times \{\text{L,N,R}\}^k$ (here L,N,R are just meaningless symbols).[1]
- A *head*, which is in a state and in a position in each tape at all times. The head reads the entries of the tapes it can see and
  - rewrites the entries with the values given by its state;
  - changes its state according to the rule;
  - moves left, right, or not at all in each tape according to which values of L, N, R the state dictated.
- One tape is designated as the input tape and never rewritten. Another is the output tape (which can be changed).
- All tapes other than the input tape start full of zeroes (I am assuming $0 \in A$.) Also the head starts at state S_{init}
- If the machine reaches S_{halt} it stops, and if the input is x and the output y, we say the machine computed y given x.

There are some variants to this definition. We can, for example assume that $A = \{0,1\}$, that $k = 1$ (with a different convention for input/output tapes), the tapes are 2-sided, etc., etc.

[1] If it bothers you that S contains functions defined on S (since this is technically a circular definition) just imagine S being the set $[1, n]$ for some n and the states are defined on $[1, n]$.

1 Some Complexity Classes

Definition 1.1 (Polynomial time)**.** The complexity class P consists of all Boolean functions $f\colon \{0,1\}^* \to \{0,1\}$ such that there exists a Turing machine T and a polynomial p such that for every $x \in \{0,1\}^*$ we have that T computes $f(x)$ in at most $p(|x|)$ steps, where $|x|$ is the length of x.

For example, consider the problem STCON, whose input is a directed graph and two vertices s, t, and whose output is 1 if there is a directed path from s to t and 0 otherwise. This problem is known to belong to P, and this is not hard to see. Starting at s, compute the vertices that are reachable with a path of length 1, then do length 2,3, and so on. This terminates after at most $\binom{n}{2} = \frac{1}{2}n(n-1)$ steps where n is the number of vertices of G. Independently of how you represent the graph (via adjacency matrix or otherwise) the algorithm will run in polynomial time, so STCON is in P.

There is a variant of P called NP, short for non-deterministic polynomial time. An example for something in NP is the following computational problem.

- Input: A graph G.
- Output: 1 if G contains a Hamilton cycle, 0 otherwise.

A nondeterministic polynomial algorithm will run as follows. First, pick a vertex. Then randomly choose neighbours of that vertex and repeat (and at the end try to come back to the original vertex). This is nondeterministic since I didn't specify which neighbours to choose.

More formally a *nondeterministic Turing machine* is a Turing machine which has two transition functions, and at each step it applies one or the other. We say a nondeterministic Turing machine computes a Boolean function f if for all $x \in \{0,1\}^*$ we have that $f(x) = 1$ iff there is a sequence of choices of transition functions that leads to output 1 when input is x.

Definition 1.2 (Nondeterministic polynomial time)**.** The complexity class NP is the class of Boolean functions computable in polynomial time by a nondeterministic Turing machine.

Proposition 1.3. *A Boolean function f is in NP if and only if there is a polynomial p and a function $g \in P$ such that for all $x \in \{0,1\}^*$ we have that $f(x) = 1$ iff there exists $y \in \{0,1\}^*$ with $|y| = p(|x|)$ such that $g(x, y) = 1$.*

Proof. First suppose f is in NP and let T be a nondeterministic Turing machine computing f in polynomial time. We can construct a deterministic Turing machine T' such that it takes input x and y and outputs what T would've outputted with x as an input, with choices of transition functions encoded in y (we don't need two input tapes for this since we can, for example, agree that even positions are supposed to be x and odd positions are y). Let g be the function computed by T'. Then it is not hard to see that $g \in P$ and we can pick y so that $g(x, y) = 1$ iff $f(x) = 1$ subject to the conditions in the size of y.

On the other direction, suppose we are given g and p. Let T' be the Turing machine computing g in polynomial time. We can reverse the above process by constructing a nondeterministic Turing machine that, given x, tries to write down the corresponding y randomly and then computing $g(x, y)$. As $|y|= p(|x|)$ and $g \in \mathrm{P}$ we see that this new Turing machine computes f in polynomial time. $\square$

Corollary 1.4. *$P \subseteq NP$.*

The major open problem in theoretical computer science is whether P=NP. This is probably not true.

Definition 1.5 (co-NP)**.** The complexity class co-NP consists of Boolean functions f such that $\neg f := 1 - f \in \mathrm{NP}$.

Alternatively, $f \in$ co-NP iff there is a polynomial p and some $g \in \mathrm{P}$ such that for all $x \in \{0,1\}^*$ we have that $f(x) = 1$ iff for all $y \in \{0,1\}^{p(|x|)}$ we have $g(x, y) = 1$. For example, testing whether a number is composite is both in NP and co-NP.

Now we arrive to something known as the polynomial hierarchy.

Definition 1.6 (Polynomial hierarchy)**.** Define Σ_0^P and Π_0^P to be P. Assuming that Σ_k^P and Π_k^P have been defined, we say that for Boolean functions f:

- $f \in \Sigma_{k+1}^P$ if and only if there exists a polynomial p and some $g \in \Pi_k^P$ such that $f(x) = 1$ iff there exists $y \in \{0,1\}^{p(|x|)}$ with $g(x, y) = 1$.
- $f \in \Pi_{k+1}^P$ if and only if there exists a polynomial p and some $g \in \Sigma_k^P$ such that $f(x) = 1$ iff for all $y \in \{0,1\}^{p(|x|)}$ with $g(x, y) = 1$.

We define $\mathrm{PH} := \bigcup_{k=0}^{\infty} \Sigma_k^P \cup \Pi_k^P$.

For example Σ_1^P is nothing but NP and Π_1^P is co-NP.

Proposition 1.7. *If P=NP, then P=PH.*

Proof. Note that if P=NP then P=co-NP (negate the function and compute it in polynomial time). If $f \in \Sigma_{k+1}^P$ then there is some $g \in \Pi_k^P$ and a polynomial p such that $f(x) = 1$ iff there is some $y \in \{0,1\}^{p(|x|)}$ such that $g(x, y) = 1$. By induction $g \in \mathrm{P}$ and Proposition 1.3 says that f is in NP=P. The proof for Π_{k+1}^P is similar. $\square$

Next, we define a complexity class which is quite different to the ones we have defined before.

Definition 1.8 (Polynomial space)**.** The class PSPACE consists of Boolean functions that can be computed by a Turing machine which uses only a polynomial amount of tape (no restriction on the number of steps).

Proposition 1.9. *NP $\subseteq$ PSPACE.*

Proof. First assume that P $\subseteq$ PSPACE. Then if f is in NP there is some g in PSPACE and a polynomial p such that $f(x) = 1$ iff there is some $y \in \{0,1\}^{p(|x|)}$ such that $g(x, y) = 1$. But then we can build a Turing machine that, given an input x does a brute-force search on $y \in \{0,1\}^{p(|x|)}$ and computes $g(x, y)$. If we erase y's that don't work and reuse the space then this clearly only takes a polynomial amount of tape to do.

It only remains to show that P $\subseteq$ PSPACE. But if a function can be computed in polynomial time then the corresponding Turing machine only reads and writes on a polynomial amount of tape (!). $\square$

We now leave the world of polynomial machines to introduces another class.

Definition 1.10 (Exponential time)**.** The class EXPTIME consists of Boolean functions that can be computed in time $\exp(O(n^k))$ for some k (where n is the size of the input).

Proposition 1.11. *PSPACE $\subseteq$ EXPTIME*

Proof. Given a Turing machine T in the middle of a computation, define its *configuration* to be its state, its position on each tape, and the values in all the cells on the tapes.

Let x be an input of size n. If T uses only a polynomial amount of space $p(n)$, has k tapes, has states S, and works in an alphabet A, then the number of possible configurations is $|S| \times (p(n))^k \times |A|^{kp(n)}$. If T goes on for longer than that amount of time then, by the pigeonhole principle, its configuration repeats and hence it is eventually periodic so it doesn't halt. Thus if T computes a function in PSPACE we see that it must do so in exponential time. $\square$

We can run the same constructions as we did before with polynomial machines.

Definition 1.12 (Nondeterministic exponential time)**.** The class NEXPTIME consists of all Boolean functions that can be computed in by a nondeterministic Turing machine in exponential time. Equivalently, f is in NEXPTIME iff there exists a function g in EXPTIME such that for all $x \in \{0,1\}^*$ we have that $f(x) = 1$ iff there is some $y \in \{0,1\}^*$ with $|y| = \exp(O(|x|^k))$ for some k such that $g(x, y) = 1$.

Definition 1.13 (Exponential space)**.** A function is in EXPSPACE if there is a polynomial p such that for all inputs of size n the function can be computed using at most $\exp(p(n))$ space.

The following is known but we do not know whether any of the inclusions are equalities (and these are all major open problems).

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{NEXPTIME} \subseteq \mathrm{EXPSPACE}.$$

2 Circuit complexity

A *circuit* is a directed acyclic graph (DAG) such that each vertex is labelled an input, an AND gate, an OR gate, or a NOT gate. An input is a vertex of in-degree 0. A NOT gate has to have in degree 1. Vertices of in-degree greater than 1 are either AND gates or OR gates (but not both).

Vertices of out-degree 0 are called outputs. Using the obvious rules, we have a well-defined function $\{0,1\}^I \to \{0,1\}^O$ where I is the set of inputs and O is the set of outputs.

If every AND and OR gate as in-degree less than or equal to some k we say that the circuit is of fan-in less than or equal to k. Often we restrict to circuits with fan-in less than or equal to 2.

Definition 2.1 (Straight-line computation)**.** Let $f\colon \{0,1\}^* \to \{0,1\}$. A *straight-line computation* of f is a sequence of functions $f_1, \dots, f_m$ such that if $x = (x_1, \dots, x_n)$ then $f_i(x) = x_i$ for all $1 \le i \le n$ and for $i > n$ we have either

- $f_i = \min\{f_{j_1}, \dots, f_{j_k}\}$ for some $j_1 \dots, j_k < i$; or
- $f_i = \max\{f_{j_1}, \dots, f_{j_k}\}$ for some $j_1 \dots, j_k < i$; or
- $f_i = 1 - f_j$ for some $j < i$,

and such that $f_m = f$. Here m is referred to as the *length* of the computation.

Clearly circuits and straight-line computations are equivalent concepts.

Lemma 2.2. *Every function $f\colon \{0,1\}^n \to \{0,1\}$ can be computed in a circuit of size at most exponential in n.*

Proof. MISSING □

Proposition 2.3. *Let f be a function that can be computed by a k-tape Turing machine T in a time $t(n)$ for inputs of size n. Then there is a family (C_n) for circuits such that $|C_n| = O(t(n)^{k+2})$ and C_n computes f for inputs of size n.*

Proof. Let $S = \{s_1, \dots, s_r\}$ be the set of states of T, and assume that the alphabet is $\{0,1\}$. Then we can encode the configuration of T at time t as follows.

- For $1 \le i \le r$ define $\sigma_i(t)$ to be 1 if T is in state s_i at time t and 0 otherwise.
- For $1 \le i \le t(n)$ and $1 \le h \le k$ define $\pi_i^h(t)$ to be 1 iff the head is at position i on tape h at time t.
- For $1 \le i \le t(n)$ and $1 \le h \le k$ define $v_i^h(t)$ to be the value in cell i of tape h at time t.

Let τ denote the transition function of T. Note that $\sigma_i(t) = 1$ iff there exists $1 \leq j \leq r$ and some $i_1, \ldots, i_k$ such that $\sigma_j(t-1) = 1$ and $\pi^h_{i_h}(t-1) = 1$ for all $1 \leq h \leq k$ and

$$\tau(s_j, v^1_{i_1}(t-1), \ldots, v^k_{i_k}(t-1))$$

has state component s_i.

Suppose we are given $1 \leq i_1, \ldots, i_k \leq t(n)$ to be the position of the head in the k tapes and j the state number we are in. To compute the next state we need to compute a function on $k+1$ variables which by Lemma 2.2 we can do with a circuit of size exponential in k, i.e. a constant time in terms of n.

It follows that we can calculate $\sigma_i(t)$ in terms of the previous configuration with a circuit of size $O(t(n)^k)$ by just searching through all of the possible $i_1, \ldots, i_k$.

Similarly we can calculate π and v with circuits of size $O(t(n)^k)$ each. So, we can compute the configuration at time t from the configuration at time $t-1$ with a circuit of size $O(t(n)^{k+1})$ so we can compute the configuration at all times with a circuit of size $O(t(n)^{k+2})$. □

With this result, we can define yet another complexity class.

Definition 2.4 (P/poly)**.** The complexity class P/poly is defined by any of the following three equivalent conditions.

1. f is in P/poly iff there is a family (C_n) of polynomial-sized circuits such that C_n computes $f(x)$ when $|x| = n$.
2. f is in P/poly iff there is a polynomial p and a sequence y_n with $|y_n| = p(n)$ and a function g in P such that
$$f(x) = 1 \iff g(x, y_{|x|}) = 1.$$
3. f is in P/poly iff there is a sequence (T_n) of Turing machines a polynomial p such that T_n has at most $p(n)$ states and T_n computes $f(x)$ when $|x| = n$.

A sequence (C_n) of circuits is *P-uniform* if there is a polynomial time algorithm that given n it generates C_n (encoded in a suitable way).

Proof of equivalence.

(1) ⇒ (2) Let y_n be an encoding of C_n and let $g(x, y) = 1$ iff the circuit encoded by y outputs 1 with input x.

(2) ⇒ (1) Using Proposition 2.3 let C'_n be a circuit computing g such that C'_n has polynomial size. Let C_n be C'_n but with the last $p(n)$ inputs restricted to y_n.

(2) ⇒ (3) Fix some n and let T compute g. Define T_n be a Turing machine that prints out y_n and then uses T to compute $g(x, y_n)$.

(3) ⇒ (2) Let y_n be an encoding of T_n and let $g(x, y) = 1$ iff the Turing machine encoded by y outputs 1 with input x.

□

3 Search and decision problems

Let g be a Boolean function of two variables. Then for any given x we get the decision problem "Does there exist a y such that $g(x, y) = 1$" and the corresponding search problem of finding such a y if it exists. A solution to the search problem is an algorithm that outputs y if it exists.

Proposition 3.1. *Suppose $P = NP$ and let f be such that there exists a g in P and a polynomial p such that for all $x \in \{0, 1\}^*$ we have $f(x) = 1$ iff there exists a $y \in \{0, 1\}^{p(|x|)}$ with $g(x, y) = 1$ (that is, f is in NP). Then there is a polynomial-time algorithm that computes a function $h: \{0, 1\}^* \to \{0, 1\}^*$ such that if $f(x) = 1$ then $g(x, h(x)) = 1$.*

Proof. For each i let g_i be the function that takes as input x and u_i where $|u_i|= i$ and outputs 1 iff there is some v with $|v|= p(|x|) - i$ such that $g(x, u, v) = 1$. Clearly all g_i's are in NP

Now run the following procedure. Start by calculating $g_1(x, 1)$ in polynomial time, which is possible since P=NP and let $u_1 = g_1(x, 1)$. Continue this process and we obtain $u = (u_1, \dots, u_{p|x|})$ such that $g(x, u) = 1$. □

Lemma 3.2. *Suppose NP is contained in P/poly and let f be such that there exists a g in P and a polynomial p such that for all $x \in \{0, 1\}^*$ we have $f(x) = 1$ iff there exists a $y \in \{0, 1\}^{p(|x|)}$ with $g(x, y) = 1$ (that is, f is in NP). Then there is a polynomial-sized family of circuits (C_n) such that if $|x|= n$ then C_n with input x computes y such that $g(x, y)$.*

Proof. Fix some n. Note that g_i, as in the previous proof, is in NP and hence in P/poly. Thus there are polynomial-sized circuit C'_i that computes g_i.

Now put together the circuits $C'_1, \dots, C'_{p(n)}$ as follows. Let C'_1 take the input $x_1, \dots, x_n, 1$ and output u_1. Then C'_2 takes inputs $x_1, \dots, x_n, u_1, 1$ and outputs u_2. Continue all the way to $C'_{p(n)}$. Call this new circuit C_n and we are done. □

Theorem 3.3 (The Karp-Lipton theorem)**.** *If $NP \subseteq P/poly$ then $\Sigma_2^P = \Pi_2^P$ (and therefore $PH = \Sigma_2^P = \Pi_2^P$).*

Proof. Let f be in Π_2^P and let h in P be such that $f(x) = 1$ iff for all y there is some z such that $h(x, y, z) = 1$ (where y, z are of appropriate polynomial-size depending on x). Define $g(x, y)$ to be 1 iff there exists some z (whose size depends polynomially on $|x|$) such that $h(x, y, z) = 1$. Clearly g belongs to NP, so by hypothesis and Lemma 3.2 there is a circuit family (C_n) of polynomial size such that for all x, if $|x|= n$ and $g(x, y) = 1$, then $h(x, y, C_n(x, y)) = 1$.

It follows that $f(x) = 1$ implies that there exists some C_n for all y such that $h(x, y, C_n(x, y)) = 1$. Conversely, if $f(x) = 0$ then there exists some y such that for all z we have $h(x, y, z) = 0$ just by definition of h. Therefore $f \in \Sigma_2^P$. To show the reverse implication $\Sigma_2^P \subseteq \Pi_2^P$ just replace f by $1 - f$. □

Lemma 3.4. *For every k there is a Boolean function $f: \{0, 1\}^* \to \{0, 1\}$ that can be computed by a circuit family (C_n) of circuits of size n^{k+1} but not by a family of circuits of size n^k.*

Proof. ES1 □

Theorem 3.5 (Kannan)**.** *For every k there is a Boolean function $f \in \Sigma_4^P$ that cannot be computed by a circuit family of circuits of size n^k.*

Proof. For n sufficiently large, Lemma 3.4 gives us some $f'_n: \{0,1\}^n \to \{0,1\}$ that can be computed by a circuit of size n^{k+1} but not by a circuit of size n^k. Choose a sensible ordering on circuits of size at most n^{k+1}. Let $f_n(x): \{0,1\}^n \to \{0,1\}$ be $C_n(x)$, where C_n be the first circuit in this ordering such that $|C_n| \leq n^{k+1}$ and no circuit of size less than or equal to n^k computes the same f_n as C_n.

Then let $f = (f_n)^\infty$. Then if $|x| = n$, we have that $f(x) = 1$ iff there exists a circuit C_n such that for all circuits D with $|D| \leq n^k$ there exists a y with $C_n(y) \neq D(y)$ and for all circuits E with $E < C_n$ there exists some circuit F with $|F| \leq n^k$ with property that for all z we have $E(z) = F(z)$ and $C_n(x) = 1$. This shows that $f \in \Sigma_4^P$. □

Corollary 3.6. *For every k there is a function $f \in \Sigma_2^P \cap \Pi_2^P$ that cannot be computed by a circuit family of size n^k.*

Proof. If NP is contained in P/poly the just combine the Karp-Lipton theorem with Theorem 3.5. If NP is not contained in P/poly we get the stronger result that there is some $f \in$ NP that cannot be computed by any circuit family of polynomial size. □

Now we define the class of logarithmic space. As $n > \log n$ we need to give the Turing machine the ability to read the whole input in the first place.

Definition 3.7 (Logarithmic space)**.** A function f belongs to the complexity class L iff there is a Turing Machine that computes f with a read-only input tape, and a work tape of size $O(\log n)$ for inputs of size n.

Definition 3.8 (Nondeterministic logarithmic space)**.** We say f belongs to the class NL iff there is a Turing Machine with a read-only input tape, a work tape of size $O(\log n)$, and a read-once certificate tape (in which the head can only stay still or move to the right) such that $f(x) = 1$ iff there is some y of polynomial size in n such that y can be put in the certificate tape and then T outputs 1 when inputted x.