Complexity and Randomness Notes

Emmett Neyman

April 2019

1 Time Complexity

Definition: Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Definition: **P** is the class of languages that are decidable in polynomial time on a deterministic single tape Turing machine. In other words,

$$P = \bigcup_k TIME(n^k)$$

Definition: A **verifier** for a language A is an algorithm V, where

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

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w. A language A is polynomially verifiable if it has a polynomial time verifier.

NP is then the class of languages that have polynomial time verifiers, which leads us to:

Theorem: A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof Idea (full proof Sipser p. 294)
We can show how to convert a pollynomial time verifier to an equivalent polynomial time NTM and vice versa - the NTM simulates the verifier by guessing the certificate, and the verifier simulates the NTM by using the accepting branch as the certificate.

Finally, we can formally define NP as follows:
Definition: **NP** is the class of languages that are decidable in polynomial time on a nondeterministic single tape Turing machine. In other words,

$$NP = \bigcup_k NTIME(n^k)$$

Some famous and common problems in NP include $CLIQUE$ (the language of all graphs with clique of some size), $SUBSET - SUM$ (the language of sets of numbers that have subsets that sum to some number), etc. This leads to the large open question of whether $P = NP$; clearly $P \subseteq NP$, but nothing else is known yet.
Definition: Language A is ***polynomial time reducible*** to language B, written $A \leq_P B$, if a polynomial time computable function exists, where for every w

$$w \in A \iff f(w) \in B$$

The function f is called the ***polynomial time reduction*** of A to B.

Theorem: If $A \leq_p B$ and $B \in P$, then $A \in P$. (full proof Sipser p. 301)

And finally, this leads naturally a definition of $NP - complete$:

Definition: Language A is ***NP-complete*** if 1. A is in NP, and 2. every language B in NP is polynomial time reducible to A.

2 Space Complexity

Definition: Let M be a deterministic TM that halts on all inputs. The *space complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of tape cells used by M on any input of size n. If M is a non-deterministic TM, then the *space complexity* $f(n)$ of M is defined to by the maximum number of tape cells that M uses across all of its branches on any input of size n.

Similarly to time complexity, we use asymptotic notation to define *space complexity classes*.

$$\text{SPACE}(n) = \{L \mid L \text{ is a language decided by a } O(f(n)) \text{ deterministic TM}\}$$
$$\text{NSPACE}(n) = \{L \mid L \text{ is a language decided by a } O(f(n)) \text{ non-deterministic TM}\}$$

Savitch's Theorem: For any function $f : \mathbb{N} \to \mathbb{R}^+$ where $f(n) \geq n$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f(n)^2)$.
Definition:

$$\text{PSPACE} = \bigcup_{k \in \omega} \text{SPACE}(n^k)$$

$$\text{NPSPACE} = \bigcup_{k \in \omega} \text{NSPACE}(n^k)$$

As a corollary to Savitch's Theorem, we have that PSPACE = NPSPACE. We also see that if a halting deterministic TM runs in $f(n)$ space, it runs in upper-bounded $2^{O(f(n))}$ time. So far, we have that

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}$$

3 Probabilistic Complexity Classes

Definition A *probabilistic Turing machine* M is a type of non-deterministic Turing machine in which each non-deterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M's computation tree on input w as follows.

$$\Pr(b) = 2^{-k}$$

where k is the number of coin-flip steps on branch b. We say

$$\Pr(M \text{ accepts } w) = \sum_{b \in B_A} \Pr(b)$$

where B_A is the set of accepting branches in M on input w. In other words, the probability that M accepts on w is the probability that we would reach an accepting state during M's computation. We define the probability that M rejects on w to be $1 - \Pr(M \text{ accepts } w)$.

Definition For $0 \leq \epsilon \leq \frac{1}{2}$, we say that M decides the language A with *error probability* ϵ if

1. $w \in A$ implies $\Pr(M \text{ accepts } w) \geq 1 - \epsilon$.

2. $w \notin A$ implies $\Pr(M \text{ rejects } w) \geq 1 - \epsilon$

Definition **BPP** is the class of languages that are decidable by a probabilistic polynomial Turing machine with error probability $\frac{1}{3}$. That is, for any $w \in A$, M accepts w with probability $\geq \frac{2}{3}$ and for any $w \notin A$, M accepts w with probability $\leq \frac{1}{3}$.

The **amplification lemma** lets us make this error probability arbitrarily small. Formally, if ϵ is strictly between 0 and $\frac{1}{2}$, then for any polynomial $p(n)$ a probabilistic polynomial Turing machine M that has error probability ϵ has an equivalent probabilistic polynomial Turing Machine M' that has error probability $2^{-p(n)}$.
Proof Idea: The general idea of the proof is to construct M' such that it runs M a polynomial number of times and then accepts if a majority of the runs result in accept states and rejects if a majority of the runs result in reject states.
Proof: Given a TM M that decides a language with error probability $\epsilon < \frac{1}{2}$ and a polynomial $p(n)$, we will construct a TM M' that decides that language with error probability $2^{-p(n)}$. We construct M' as follows. On input x, it will first calculate a value k as we will explain later, then run x on M $2k$ times. Finally, M' accepts iff a majority of the $2k$ runs accept. Now it remains to calculate k so that the probability that M' incorrectly accepts or rejects is appropriately bounded. M' is incorrect only when at least half of the simulated runs of M are incorrect. So, we want to bound the probabilty that at least half of all the simulations give an incorrect answer. Consider some sequence S of simulated runs of M and let P_S be the probabilty that M' produces those runs when it simulates M. We will say that S has c correct runs and w incorrect runs ($c + w = 2k$). In other words, M' will output the wrong answer when $w \geq c$. If this is the case, we say that S is *bad*. Let's call P_x the probability that M is wrong on input x. Then, if S is *bad*,

$$P_S \leq (P_x)^w (1 - P_x)^c \leq (\epsilon)^w (1 - \epsilon)^c \leq (\epsilon)^k (1 - \epsilon)^k$$

If we sum P_S for all *bad* sequences P, we will get the probability that M' outputs the wrong answer. Since there are 2^{2k} possible sequences, we get that the probability that M' outputs incorrectly is

$$\sum_{bad\ S} P_S \leq 2^{2k} (\epsilon)^k (1 - \epsilon)^k = (4\epsilon(1 - \epsilon))^k$$

Since $\epsilon < \frac{1}{2}$, $(4\epsilon(1 - \epsilon))^k < 1$ and we have that M''s error probability decreases exponentially with respect to k. All that remains is to set k to a specific value such that M' has error probability $2^{-p(n)}$. If we let $\alpha = -\log_2(4\epsilon(1 - \epsilon))$, then we can set $k \geq \frac{p(n)}{\alpha}$ to optain a TM M' that accepts the same language as M with error probability $2^{-p(n)}$.

We know that P $\subseteq$ BPP but we don't know anything about the relationship between BPP and NP.

Definition Let **RP** be the class of languages that are decided by a probabilistic polynomial Turing machine where inputs that are in the language are accepted with a probability of at least $\frac{1}{2}$ and inputs that are not in the language are rejected with probability 1. That is, if we the Turing machine accepts, we know the input must be in the language.
We can think of the class RP as the class of problems for which there exists a *Monte Carlo algorithm*. That is, if the algorithm accepts, we know it's correct. But if the algorithm rejects, it's possible that it actually should have accepted.

We know that RP $\subseteq$ BPP, P $\subseteq$ RP, and RP $\subseteq$ NP. To see why P $\subseteq$ RP, we notice that a deterministic poly-time algorithm is just a stronger type of *Monte Carlo algorithm*, one that ignores all coin flips and makes its decisions deterministically. To see why RP $\subseteq$ NP, we notice that a *Monte Carlo algorithm* for a given language is just a special type of non-deterministic algorithm for that language.

Definition Let **coRP** be the complement of the class RP. That is, coRP is the class of languages that have probabilistic algorithms that always accepts when a string is in the language but sometimes gives the wrong answer when a string is not in the language. This means that if the Turing Machine rejects, we know the input is not in the language.

Definition We define **ZPP** as RP $\cap$ coRP. Intuitively, ZPP contains all the languages that have two types

of randomized algorithms: one that has no false-positives and one that has no false-negatives.
An alternative definition of ZPP is as follows. Consider a non-deterministic, polynomial time Turing machine that has three outputs: accept, reject, and ?. This machine must output the correct answer with probability at least $\frac{2}{3}$ and can never output an incorrect answer. On any input, this machine can output ? with probability at most $\frac{1}{3}$. The set of all languages recognized by these Turing machines is the class ZPP.

Definition Consider a non-deterministic, polynomial time Turing machine that, accepts a string x iff a majority of its branches of computation accept x. The set of languages that are recognized by such a Turing machine is the class **PP**, for probabilistic polynomial time. That is, a language is in PP if it is accepted by a probabilistic polynomial Turing machine with error probability less than $\frac{1}{2}$.

Theorem: NP $\subseteq$ PP.
Proof: Consider a language $L \in$ NP that is decided by a non-deterministic TM M. We will construct a probabilistic TM M' that decides L by majority (as described above). M' is identical to M except it has a new initial state and, from that state, makes a non-deterministic choice that decides how its computation should proceed. Either, it can choose to do the same computation that M does or it can choose to do a computation that always accepts and takes the same number of steps that M would take. Consider an arbitrary input w. M computes an output on w in $p(|w|)$ steps and will have $2^{p(|w|)}$ computations. Thus, M' will have $2^{p(|w|)+1}$ computations since we added a new initial state and a new computation step. We see that at least half of these computations will end up accepting w since half of the computations will take the branch that always accepts. Thus, a M' will accept w a majority of the time iff there is at least one branch of computation of M that accepts w. This is the case only when $w \in L$. Thus, NP $\subseteq$ PP.

Theorem: PP $\subseteq$ PSPACE.
Proof Idea: Let M be a probabilistic polynomial time TM. We can construct a deterministic TM M' that accepts the same language M as follows. On input w, M' simply tries each of the computation branches and accepts if a majority of them accept. Each of the branches takes polynomial space and that space can be reused for each of the computation branches.
An important differnce between PP and BPP is that in PP, the error probability can be a function of the length of the input. In BPP however, it must be independent of the lengh of the input.

We now have that ZPP $\subseteq$ RP $\subseteq$ BPP $\subseteq$ PP.

4 References

- CIS 511 Notes by Emmett Neyman and Olek Gierczak
- *Introduction to the Theory of Computation, 3rd Edition* by Michael Sipser
- *Computational Complexity* by Christos Papadimitriou
- *Theory of Computational Complexity* by Ding-Zhu Du, Ker-I Ko, and Ker-I Ko

BPP, NP, and the Polynomial Hierarchy

Emmett Neyman

April 2019

BPP in P/poly

Adleman's Theorem: All languages in BPP have polynomial circuits. That is, BPP $\subseteq$ P/poly.

Proof: We will show that any language $L \in$ BPP has a polynomial family of circuits $C_0, C_1, C_2, ...$ that accepts L. The structure of our proof will show that for n, where n is the length of some string w, there is a polynomial circuit C_n that accepts w iff $w \in L$. Note that this construction can't be that straightforward; if it were, then we would have P = BPP. So, this proof will contain a step that is not "efficiently constructive".

Let's consider a sequence of bit strings $A_n = (a_1, a_2, ..., a_m)$ where $a_i = \{0,1\}^{p(n)}$ for $i = 1, ..., m$ where n is the length of the input, $m = 12(n+1)$, and $p(n)$ is the length of the computation for a string of length n on a non-deterministic probabilistic Turing machine M that decides L. Intuitively, we can think of each bit string in A_n as a "path" of computation through the machine M for any input of length n where each bit represents the choice M makes at that step. The circuit C_n, on input w of length n, will simulate M for each of the m paths in A_n and then take the majority decision. Since we can create a circuit that simulates a Turing machine computation, we know we can construct a circuit C_n given a sequence A_n. Thus it only remains to show that there is a sequence A_n that works correctly.

That is, we want to show that, for all $n > 0$, there is a set of sequences $A_n = (a_1, a_2, ..., a_m)$ of bit strings such that less than half of them are *bad*. Here, a bit string is *bad* if the computation path it represents leads to either a false positive or a false negative: in either case, the wrong answer. Consider a sequence A_n of m $p(n)$ length bit strings obtained by sampling randomly from $\{0,1\}^{p(n)}$. Consider the following question: what is the probability that for each input of length n, A_n gives the correct answer more than half the time? If this is the case, we know that A_n is correct since we take the majority decision of each of the m branches. We will show that this probability is $\geq \frac{1}{2}$.

From the definition of BPP, we know that for a given input w of length n, at most $\frac{1}{4}$ of the computations are wrong (we originally defined BPP as having error probability $\frac{1}{3}$ but we could also have defined it as having error probability $\frac{1}{4}$ thanks to the amplification lemma). Since the bit strings in A_n were selected randomly, we expect $\frac{m}{4}$ *bad* bit strings. Using the Chernoff bound (fancy algebra/arithmetic), we see that the probability of getting $\frac{m}{2}$ or more *bad* bit strings is $e^{-\frac{m}{12}}$ which is less than $\frac{1}{2^{n+1}}$. Since this probability is for a single input w, we need to sum over all possible inputs to find the probability that there is an input w that isn't accepted correctly by A_n. Thus, we get $2^n \cdot \frac{1}{2^{n+1}} = \frac{1}{2}$. So, we see that the probability of picking a sequence A_n that works for all inputs n is at least $\frac{1}{2}$.

We'll notice though that even though we've shown such a sequence A_n exists, we did not show how we could find it; in other words, we didn't give any hints about the relationship between P and BPP. Now that we've shown that such an A_n exists, we know we can create a circuit C_n with polynomial many gates with respect to n. This circuit will take the majority decision of the m simulations and will accept a string w iff $w \in L \cap \{0,1\}^n$.

BPP in the Polynomial Hierarchy

Sipser–Gács–Lautemann Theorem: BPP is in the polynomial hierarchy. More specifically,

$$\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$$

Proof: We will show that BPP $\in \Sigma_2^P$ and, since BPP is closed under complement, it will follow immmediately that BPP $\subseteq \Sigma_2^P \cap \Pi_2^P$. Consider a language $L \in$ BPP. We know that there is a non-deterministic probabilistic Turing machine M that uses $p(n)$ computations and decides L with low error probability. Let w be some string of length n and let $A(w) \subseteq \{0,1\}^{p(n)}$ be the set of computation paths that accept w. We know that if $w \in L$ then $|A(w)| \geq 2^{p(n)}(1 - \frac{1}{2^n})$ and if $w \notin L$ then $|A(w)| \leq 2^{p(n)}\frac{1}{2^n}$ by the amplification lemma.

Let $a, b \in \{0,1\}^{p(n)}$ be two bit strings. We define $a \oplus b$ to be the bitwise XOR of a and b. We see that $a \oplus b = c$ if and only if $c \oplus b = a$. That is, the function $\oplus y$, for a bit string y of length $p(n)$, is a function that when applied twice to another bit string, gets that same bit string back. We also note that if r is a random bit string obtained by flipping an unbiased coin $p(n)$ times, then $y \oplus r$ is also a random bit string even if y is not random. Let t be some bit string of length $p(n)$. We consider the set $A(w) \oplus t = \{a \oplus t \mid a \in A(w)\}$ which we call the *translation* of $A(w)$ by t. Notice that $|A(w)| = |A(w) \oplus t|$ for any string t.

The proof will proceed by showing the following fact: if $w \in L$ then $A(w)$ is sufficiently large enough to have a small set of translations cover the entirety of $\{0,1\}^{p(n)}$ and if $w \notin L$, then $A(x)$ is so small that no set of translations can cover the entirety of $\{0,1\}^{p(n)}$.

More formally, consider $w \in L$ and consider the sequence $T = (t_1, t_2, ..., t_{p(n)})$ of $p(n)$ many bit strings of length $p(n)$. We obtain T by flipping $p(n)$ unbiased coins. For a fixed bit string $b \in \{0,1\}^{p(n)}$, we say that T covers b if $b \in A(w) \oplus t_i$ for some $i \leq p(n)$. Notice that $b \in A(w) \oplus t_i$ iff $b \oplus t_i \in A(w)$. We see that since $w \in L$, the probability that $b \notin A(w) \oplus t_i$ is $\frac{1}{2^n}$. Therefor, the probability that b is not covered by T is $(\frac{1}{2^n})^{p(n)} = 2^{-np(n)}$. So, the probability that there is a bit string of length $p(n)$ that is not covered by T is $2^{p(n)} \cdot 2^{-np(n)} = 2^{-(n-1)p(n)}$. So, we have shown that a random sequence of $p(n)$ bit strings of length $p(n)$ has a very high probability of covering every bit string of length $p(n)$ and thus can conclude that such a T must exist.

Now consider $w \notin L$. In this case, $|A(w)|$ is so small (an exponentially small fraction of all bit strings of length $p(n)$) that no sequence of bit strings can cover all bit strings. Thus, there is a sequence T that covers $\{0,1\}^{p(n)}$ iff $w \in L$.

Now we are ready to show that $L \in \Sigma_2^P$. Specifically, we can define L as follows:

$$L = \{w \mid \exists T \in \{0,1\}^{p(n)^2}, \forall b \in \{0,1\}^{p(n)}, \exists i \leq p(n), b \oplus t_i \in A(w)\}$$

We see that this set must be in Σ_2^P from the order of the quantifiers: $\exists\forall$. The last existential quantifier can really by thought of as a disjunction over all the polynomial-many t_i's so does not "factor in" to the quantifier count. Thus, since $L \in \Sigma_2^P$ and L is an arbitrary language in BPP, we have that BPP $\subseteq \Sigma_2^P$ and because BPP is closed under complement, BPP $\subseteq \Pi_2^P$. Thus,

$$\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$$

The Collapse of the Polynomial Hierarchy

Karp-Lipton Theorem: If NP is contained in P/poly, then the polynomial hierarchy collapses to its second level. More formally, if NP $\subseteq$ P/poly then $\mathbf{PH} = \Sigma_2^P$. The original theorem states that if the language SAT can be decided by a polynomial circuit family, then **PH** collapses to its second level. The following proof will consider SAT.

Proof: The proof will leverage the fact that SAT is self-reducible. This means that there is a polynomial time Turing machine M^{SAT} with the restriction that, on inputs of length n, the query string can only ever have queries of length $n-1$. In other words, there is a polynomial-time algorithm for SAT that invokes a SAT oracle on smaller instances of the problem. Self-testing is an important result of a problem being self-reducible.

Consider a family of circuits $C = (C_0, C_1, ...)$ that decide SAT. Going forward in this proof, we will allow for the machine M^{SAT} to use as its oracle an initial segment of C rather than a machine that solves SAT. In other words, we will use the machine M^{C_n} that invokes the appropriate circuit from the circuit family for whatever query is on its query string. Since we know all queries are no longer than the input, we only need an initial segment of C up to C_n. An initial segment of C, C_n, *self-tests* if it is the case that for all inputs w up to length n, $M^{C_n}(w) = C_n(w)$. This means that all strings w give the same answer when inputted to the appropriate circuit in C and when inputted to M^{C_n}. If this equality holds for all strings w, then we know that C_n is a correct initial segment of C, a polynomial circuit family for SAT.

The remainder of the proof has the following goal. Given that SAT can be solved by a polynomial circuit family, we want to show that some language $L \in \Sigma_3^P$ is also in Σ_2^P. That is, we want to show that $\Sigma_3^P = \Sigma_2^P$ and it will follow that $\mathbf{PH} = \Sigma_2^P$. Consider the language $L \in \Sigma_3^P$. We know L is of the following form:

$$L = \{x \mid \exists y \forall z (x, y, z) \in R\}$$

where R is a polynomially balanced relation decidable in NP. Since SAT is NP-complete, we know there is a reduction F from the relation R to SAT. That is, $(x, y, z) \in R$ iff $F(x, y, z) \in SAT$. Consider the input x and let $p(|x|)$ be the length of the largest possible $F(x, y, z)$. Since R is polynomially balanced and F runs in polynomial time, we know that $p(n)$ is a polynomial. Now it remains to show that $L \in \Sigma_2^P$. We claim that $x \in L$ iff the following is true:

There exists an initial segment of a circuit family C, $C_{p(|x|)}$ and there exists a string $y, |y| \leq p(|x|)$ such that for all strings $z, |z| \leq p(|x|)$ and boolean expressions $w, |w| \leq p(|x|)$, we have 1. $C_{p(|x|)}$ successfully self-tests on w, that is $M^{C_n}(w) = C_n(w)$ and 2. $C_{p(|x|)}$ outputs true on the expression $F(x, y, z)$.

Notice that the form of this definition (the quantifier alternation) places L in Σ_2^P so it remains to show that our claim is true. If the above emphasized condition is true, then we know by 1. that $C_{p(|x|)}$ is a correct initial segment of a polynomial circuit family that decides SAT. This means that is can correctly decide instances of $(x, y, z) \in R$ by deciding instances of $F(x, y, z) \in SAT$. This implies that if the above condition holds, $x \in L$. Now we show that if $x \in L$, the above condition must hold. If $x \in L$, we know that there must exist a string y such that forall strings z, $R(x, y, z)$. Because we assume that SAT can be solved by a polynomial circuit family C, we know that there must exist successfully self-testing initial segment of C. The existence of this correct initial segment implies that the above condition holds since F is a correct reduction from R to SAT.

Thus, $\Sigma_2^P = \Sigma_3^P$ and the polynomial hierarchy collapses to its second level.

References

- *Computational Complexity* by Christos Papadimitriou
- *Computational Complexity Theory* by Steven Rudich, Avi Widgerson, and Editors