CS 2401 - Introduction to Complexity Theory

Instructor: Toniann Pitassi

Lecturer: Thomas Watson

Scribe Notes by: Benett Axtell

1 Review

We begin by reviewing some concepts from previous lectures.

1.1 Basic Complexity Classes

Definition DTIME is the class of languages for which there exists a deterministic Turing machine. We define it as a language L for which their exists a Turing machine M that outputs 0 or 1 that is both correct and efficient. It is correct iff M outputs 1 on input x and efficient if M runs in the time $O(T(|x|))$. The formal definition of this is as follows:
DTIME$(T(n)) = \{L \subseteq \{0,1\}^* : \exists$ TM M (outputting 0 or 1) s.t.
$\forall x : x \in L \Leftrightarrow M(x) = 1$ and M runs in time $O(T(|x|))\}$

Definition The complexity class P is the set of decision problems that can be solved in polynomial DTIME. Formally: $P = \cup_{c \geq 1}$DTIME(n^c)

Definition NTIME is the class of languages for which there exists a non-deterministic Turing machine (meaning the machine can make guesses as to a witness that can satisfy the algorithm). We define this as:
NTIME$(T(n)) = \{L \subseteq \{0,1\}^* : \exists$ TM M (outputting 0 or 1) s.t.
$\forall x : x \in L \Leftrightarrow \exists w \in \{0,1\}^{O(T(|x|))} M(x,w) = 1$ and M runs in time $O(T(|x|))\}$

There are two definitions of NTIME. Above is the external definition where a witness is passed to the machine. The internal definition is not given a witness and can guess possible witnesses.

Definition The complexity class NP is the set of decision problems that has a solution that can be verified in polynomial time. Formally: $NP = \cup_{c \geq 1}$NTIME(n^c)

Definition NP-complete is the class of the hardest problems in NP. A language L' is NP-complete iff it is in NP and it is NP-hard, meaning that and language L in NP can be reduced to L' ($L' \in NP$ and $\forall L \in NP$ $L \leq_p L'$). This definition means that L' can only be in P if P = NP ($L' \in P \Leftrightarrow P = NP$)

1.2 Reductions

Reductions are used to prove NP-hardness and NP-completeness.

Definition A Karp (or mapping) reduction uses a polytime computable mapping function to reduce one problem to another. The mapping function maps "yes" inputs of L to yes inputs of L' and "no" inputs of L to "no" inputs of L' such that $x \in L \Leftrightarrow f(x) \in L'$.

Definition A Cook (or oracle) reduction is a more general reduction that allows the function f to make repeated queries to an oracle that can decide L'.

These two types of reductions mean that there are two definitions of NP-complete. It is generally presumed that these two sets of NP-complete are not equivalent and that there are some Cook NP-complete problems that are not Karp NP-complete.

1.3 Examples of NP-Completeness

We previously showed that both Circuit-SAT and 3SAT are NP-complete. To do this we first show that Circuit-SAT is NP-complete from first principles. Because reductions are transitive, we can show that 3SAT is NP-hard by reducing Circuit-SAT to 3SAT once we know that Circuit-SAT is NP-complete.

Theorem 1 *Ladner's Theorem*
$P \neq NP \Rightarrow \exists L \in NP$ *s.t.* $\{L \notin P$ *and* L *is not NP-complete*$\}$

It is an important distinction that not everything is either P or NP-complete. Some candidates are believed to be NP-Intermediate.

2 Examples of Karp Reductions

2.1 Review of 3SAT

3SAT is the language of all satisfiable 3CNF formulas (φ). 3CNF is a formula in conjunctive normal form (a set of ANDs of clauses) where each clause is an OR of 3 literals (a variable or its negative).
Or: $3SAT = \{3CNF\varphi : \exists w, \varphi(w) = 1\}$
An example of 3CNF would be: $(w_1 \vee \overline{w_2} \vee w_3) \wedge (w_2 \vee \overline{w_3} \vee w_4) \wedge (\overline{w_1} \vee w_3 \vee \overline{w_4})$

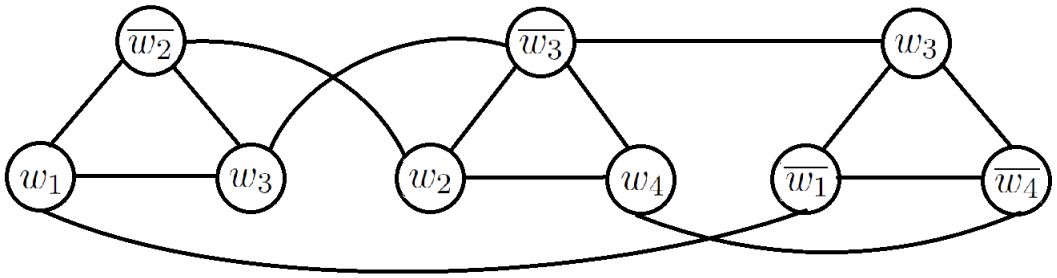
2.2 Independent Set

Definition Given a graph G and a number k, an independent set of G is a set of k pairwise nonadjacent nodes (meaning a set of nodes with no edges between them). The Independent Set problem is the language of pairs (G, k) such that G has independent set of size k.

Theorem 2 *INDSET is NP-complete*

Proof
It is trivial to show that INDSET is in NP. It is just necessary to guess a set of nodes for the given graph.
To show that INDSET is NP-hard, we will show that 3SAT $\leq_p$ INDSET. To reduce INDSET to 3SAT we need to model the 3CNF formula as a graph G with an independent set of size k where k is the number of clauses in φ. G will have a node for each occurrence of a literal in φ and an edge iff two nodes are conflicting or if they are literals in the same clause.

Using the example 3CNF formula from above, the equivalent graph for INDSET would be:



There are two steps to checking that this reduction works. First that if φ is satisfiable then G has INDSET of size k and, conversely, that φ is satisfiable if G has INDSET of size k which has exactly one node for each clause and no conflicting literals (so it is consistent with an underlying satisfying assignment to phi).
Given a satisfiable φ, we can find an independent set of the graph G by taking a satisfying assignment of φ and selecting one of its satisfied literals from each clause, and only picking either positive or negative variables. For the above example, a satisfying solution would be $\overline{w_2}$, $\overline{w_3}$, and $\overline{w_4}$.

3 Types of Computational Problems

- Decision (or Boolean) problems: Only output 0 or 1
- Search problems: Output a solution to the problem
- Optimization problems: Output (for example) the largest possible answer to the problem.

3.1 Examples Outputs for Different Computational Problems

Problem	Input	Decision	Search	Optimization
Path	(G, s, t)	$\exists$ an s-t path	Any s-t path	Shortest s-t path
3SAT	3CNF φ	$\exists$ a satisfying assignment φ	Any satisfying assignment to φ	The assignment of φ that satisfies the maximum number of clauses
INDSET	(G, k)	$\exists$ INDSET of size k	Any INDSET of size k	Largest INDSET

Theorem 3 *Only considering decision problems does not lose generality because optimization problems can be reduced to search problems and search problems can be reduced to decision problems.*
$P = NP \Rightarrow \forall L \in NP$ *and a verifier TM* M *for* $L, \exists$ *polytime TM* B: $\forall x \in L, M(x, B(x)) = 1$
In the above, TM B *outputs a witness that resolves* M *to 1.*

To prove this we will determine a witness for a search problem bit-by-bit thereby breaking it down into a series of decision problems.

Proof
$L' = \{(x, w)$ s.t. $\exists w'$ s.t. $M(x, w1w') = 1\}$ Where w' is of length = (witness length for M) $-|w|-1$.
This is seeing if w can be extended to the witness by setting the next bit to 1.
$L' \in NP$ so $L' \in P$
Next we write the algorithm for B as:

$B(x)$: $w \leftarrow \varepsilon$ (empty string)
Until $|w| =$ witness length for M:
If $(x, w) \in L'$ then $w \leftarrow w1$
Else $w \leftarrow w0$
Output w

We have an invariant that w is a prefix of some witness for x. If we are able to maintain this invariant then the algorithm is valid. It is obvious that an empty string is a prefix for the witness. When L' is true, then adding a 1 is valid and otherwise adding a 1 would be against the invariant because w would no longer be a valid prefix, so in that case adding a 0 is valid.

4 coNP

Definition coNP is the set of all languages for which all their complement language are in NP. Notice that coNP is not the complement of NP.

$coNP = \{L \subseteq \{0,1\}^*$ s.t. $\overline{L} \in NP\}$ where $\overline{L}$ is complement to L ($\overline{L} = \{0,1\}^* \backslash L$).

An alternative definition for coNP is as follows:

$L \in coNP \Leftrightarrow \exists$ polynomial p: $\mathbb{N} \to \mathbb{N}$ and a polytime TM M s.t.
$\forall x$: $x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)}, M(x, u) = 1$

Notice that this definition is the same as the definition of NP except the $\forall u$ replaces the $\exists w$. To prove that a language is in coNP, we can take the NP verifier function for the complement L and flip the bits of the witness.
As it happens, separating NP from coNP is harder than separating P and NP.

Fact $NP \neq coNP \Rightarrow P \neq NP$
The class P is closed under complement meaning that all L in P have $\overline{L}$ in P.

4.1 Example Problems in coNP

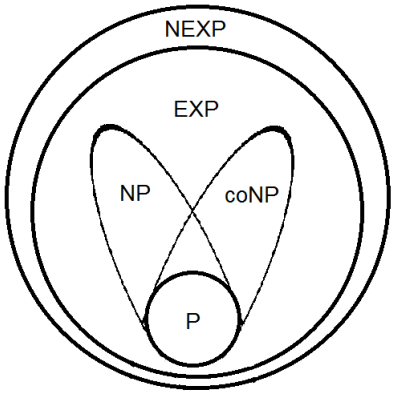
- UNSAT (complement of SAT): CNFs that are not satisfiable
- TAUT (tautology): Formulas φ where all assignments are satisfying

Both of these examples are coNP-complete which is analogous to NP-complete.

5 Exponential Time: EXP

Definition
$EXP = \cup_{c \geq 1} \text{DTIME}(2^{n^c})$
$NEXP = \cup_{c \geq 1} \text{NTIME}(2^{n^c})$

Fact $P \subseteq NP \subseteq EXP \subseteq NEXP$
This is because there are exponentially many possible witnesses for NP, and to try all possible witnesses would be in EXP.
Below is a visual representation of the relationships between these classes:



Theorem 4 $EXP \neq NEXP \Rightarrow P \neq NP$

Proof
We will prove the contrapositive (P=NP $\Rightarrow$ EXP=NEXP) by reducing EXP=NEXP to P=NP.
A TM's running time is a function of input length, so we can create a language L_{PAD} that pads the input of L to be exponentially bigger, changing what function the running time is, in terms of the input length, without changing the running time.
$L \in$ NEXP $\Rightarrow L_{PAD} \in$ NP $\Rightarrow L_{PAD} \in$ P $\Rightarrow L \in$ EXP
We will define L_{PAD} and show NEXP $\rightarrow$ NP and P $\rightarrow$ EXP implications.

Assume M is an NDTM for L running in time $O(2^{n^c})$.
$L_{PAD} = \{(x, 1^{2^{|x|^c}}) : x \in L\}$
$L_{PAD} \in$ NP because you could ignore the padding.
The algorithm for L_{PAD} is as follows: Check that the input is in the correct form (i.e: $(x, 1^{2^{|x|^c}})$), and output 0 if it is not. Otherwise run $M(x)$ for $2^{|x|^c}$ steps and output the that same answer. This runs in polytime on its input $(x, 1^{2^{|x|^c}})$ in $2^{|x|^c}$ steps.
Having shown that L_{PAD} is in NP, we can say that L_{PAD} in in P because of our assumption that P = NP. At this point, it is trivial to show that L is in EXP.

6 Time Hierarchies

Definition A time hierarchy provides a result that says a TM that has more time will have more power.

Theorem 5 *Deterministic Time Hierarchy*
$\forall$ *"time-constructible"* $f, g : \mathbb{N} \to \mathbb{N}$ *s.t.*
$f(n)logf(n) = o(g(n))$
$DTIME(f(n)) \subsetneq DTIME(g(n))$
So there is a function g that is a little bigger than a function f and there is something that can be done in g that can't be done in f.

Theorem 6 *Non-deterministic Time Hierarchy*
$\forall$ *"time-constructible"* $f, g : \mathbb{N} \to \mathbb{N}$ *s.t.*
$f(n+1) = o(g(n))$
$NTIME(f(n)) \subsetneq NTIME(g(n))$

Recall that there is a Universal TM $\mathcal{U}$ that can simulate another TM.
$\exists$ TM $\mathcal{U}$: $\forall x, \alpha \in \{0,1\}^*$
Correct: $\mathcal{U}(x, \alpha) = M_\alpha(x)$
Efficient: if M_α halts in T steps on x then $\mathcal{U}$ halts in $O(TlogT)$.
The log is the necessary overhead to simulate the TM, and the constant factor in O depends on α but not on x.

Proof
We will prove the deterministic theorem for the special case where f is linear and g is quadratic using diagonalization.
DTIME(n) $\subsetneq$ DTIME(n^2)
We define an algorithm D for the function g that on input x will run $\mathcal{U}(x, x)$ for $|x|^{1.5}$ steps. The exact value of 1.5 is not important, but it must be bigger than linear and less than quadratic to leave room for the overhead of TM simulation. This algorithm is simulating a TM on an input that is its own code. If the TM halts and outputs bit b (0 or 1) then D will output the opposite and output 0 otherwise.
$L(D)$ is the language solved by the algorithm for g. This is a contrived language to show the time hierarchy. We know that $L(D) \in$ DTIME(n^2). Now we claim that $L(D) \notin$ DTIME(n)

Consider any M running in time O(n). Let x be a bit string which is an encoding of M. This input needs to be bigger than the constant in O and long enough that (runtime of $\mathcal{U}(x, x)) = c|x|log|x| < |x|^{1.5}$. This guarantees that $\mathcal{U}$ will run to completion and it will deliberately output the opposite of the output.
$x \in L(D) \Leftrightarrow D(x) = 1 \Leftrightarrow \mathcal{U}(x, x) = 0 \Leftrightarrow M(x) = 0 \Leftrightarrow x \notin L(M)$
So M does not solve D and D cannot be solved in linear time because of $\mathcal{U}$.

7 Next Lecture

In the next lecture we will cover Space Complexity.