CS 810: Introduction to Complexity Theory 1/28/2003

Lecture 3: Complexity Classes and NP-Completeness

Instructor: Jin-Yi Cai Scribe: David Koop, Martin Hock

1 Review of Hierarchy Theorems

In the last lecture, we examined the Time-Hierarchy and Space-Hierarchy Theorems and the diagonalization that underlies their proofs. Remember that our goal was to show the existence of functions that will run in a given time or space bound but will not be computable under more constrained bounds. More specifically, we tried to show that $\mathsf{DTIME}[t(n)] \subsetneqq \mathsf{DTIME}[T(n)]$ where

$$\mathsf{DTIME}[f(n)] = \{L \mid \exists \text{ TM } M \text{ s.t. } L(M) = L,\ \text{time}_M(n) \leq f(n)\}$$

We couldn't use the requirement that $t(n) \ll T(n)$ or $t(n)/T(n) \to 0$ because of the overhead associated with more symbols and tapes. Thus, we needed to require that

$$\frac{t(n)\log n}{T(n)} \to 0$$

In addition, is important to note that $t(n)$ and $T(n)$ must *both* be time-constructible. The theorem is simply that if these conditions hold, we achieve the desired goal.

Theorem 1 (Time-Hierarchy Theorem). *If $t(n)$ and $T(n)$ are fully time-constructible and $\frac{t(n)\log n}{T(n)} \to 0$, then $\mathsf{DTIME}[t(n)] \subsetneqq \mathsf{DTIME}[T(n)]$.*

The proof is based on a delayed diagonalization argument. Thus, we try to kill all Turing machines that run in $t(n)$ time, running $T(n)$ as a subroutine as before. Consider infinitely many inputs so that M_1 runs on input x_1 then M_2 runs on x_2 then M_1 runs again but this time on input x_3. This continues as:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	$\ldots$
M_1	M_2	M_1	M_2	M_3	M_1	M_2	M_3	M_4	M_1	$\ldots$

and it should be clear that one can construct a machine in $T(n)$ which kills off every machine that runs in $t(n)$ time. The $\log n$ factor comes from simulating an arbitrary number of tapes using a fixed number of tapes (2 tapes).
The Space-Hierarchy Theorem is similar to the time version except that there is no $\log n$ fudge factor. Also, recall that if the space required is less than $\log n$, the function is uninteresting so we only really care about functions greater than this value.

Theorem 2 (Space-Hierarchy Theorem). *If $s_1(n), s_2(n) \geq \log n$ are both space-constructible and $\frac{s_1(n)}{s_2(n)} \to 0$, then $\mathsf{DSPACE}[s_1] \subsetneqq \mathsf{DSPACE}[s_2]$*

Recall that

$$\mathsf{DSPACE}[f(n)] = \{L \mid \exists \text{ TM } M \text{ s.t. } L(M) = L,\ \text{space}_M(n) \leq f(n)\}$$

The proof of this theorem is similar to time version.
While both of these theorems are important, there is something unsatisfying about them. They show the existence of a hierarchy, but there are no specific examples of these functions that can't be computed in time $t(n)$ or space $s(n)$. It is similar to proof that transcendental numbers exist. Cantor's proof showed that such numbers exist, but his proof did not give any concrete examples of such numbers. Hermite's proof that e is transcendental somehow is more satisfying. At some level, these hierarchy theorems "don't sing music to your heart." In this class, we will now attempt to study actual problems instead of concentrating on existence proofs.

2 Complexity Classes

In order to think about more problems and their relationship to complexity theory, we need to examine the major classes of this area. Recall the following definitions of the two most prominent complexity classes:

$$\mathsf{P} = \bigcup_{i>0} \mathsf{DTIME}[n^i + i]$$
$$\mathsf{NP} = \bigcup_{i>0} \mathsf{NTIME}[n^i + i]$$

where NTIME is non-deterministic time. It is harder to formally define non-deterministic time so consider the following description. In running a program, the machine can take a variety of different paths to reach an ending state. For NTIME, we only require that there is *some* path that accepts in the time bounded. There may be other paths that take much longer times, but as long as there is one that fits the bound, the language is in NTIME. Another way to think about this concept is to imagine the program trying every path simultaneously, and stopping all paths when the clock runs out (it has ticked away $f(n)$ instructions). If we reach an accept state, the language is in $\mathsf{NTIME}[f(n)]$.

It is clear that $\mathsf{P} \subseteq \mathsf{NP}$, and one of the greatest outstanding problems in computer science is whether $\mathsf{P} = \mathsf{NP}$. It is widely believed that $\mathsf{P} \neq \mathsf{NP}$, but no one has been able to show this yet.

2.1 Non-Deterministic Log Space (NL)

Another important class is *non-deterministic log space* (NL). There are two different approaches to understanding this class: the first is to formally define it in terms of a Turing Machine and the other is to think about graph accessibility. Formally, we consider a Turing Machine with two tapes: an input tape that is read-only and a work tape that can be both written to and read from. A language is in this class if the space used on the work tape is logarithmic in the length of the input tape.

$$\mathsf{NL} = \mathsf{NSPACE}[\log n]$$

The graph accessibility (or s-t connectivity) problem concerns a directed graph $G = (V, E)$ and two specified vertices s and t in V. We wish to know whether the vertices s and t are connected by a directed path from s to t. By definition, the size of the input for this problem is $|V| = n$ and $|E| = m \leq n^2$. This problem in easily solved in polynomial time (with respect to n and m) by a depth-first search. While this algorithm is fast, it requires n space to keep track of the visited vertices. We would like to find an algorithm that is more space-efficient.

2.2 s-t Connectivity

Consider the process starting at the vertex $v = s$, guesses a new edge v', and checks to see if the edge (v, v') exists. If it does we forget the edge v' and repeat the process for $v = v'$. If not, we choose a new v' and continue as before. Notice that if $v' = t$ and $(v, v') \in E$, we have found the desired path. The major problem with this algorithm is that if no path exists and there is a cycle, it will never end. However, we know that any path will be at most as long as the number of vertices n. Thus, if we run a clock, we can stop this process after n steps.

Algorithm 1: Space-Efficient Algorithm for s-t connectivity
Input: The starting vertex s, a time bound n, and the starting number of steps (0)
Output: TRUE or FALSE based on whether the algorithm finds a path

```
FINDPATH(v,n,i)
(1)   if i > n
(2)      return FALSE
(3)   Given a vertex v, guess a new vertex v'
(4)   Check the input tape to see if (v, v') ∈ E
(5)   if (v, v') ∈ E
(6)      if v' = t
(7)         return TRUE
(8)      else
(9)         FINDPATH(v',n, i + 1)
(10)  else
(11)     FINDPATH(v, n, i + 1)
```

In essence, we have a clock that ticks up to n, and for $i < n$, we guess the next vertex v', then check to see if $(v, v') \in E$. If it is not, we abort; if it is and $v' = t$ we accept; otherwise, if it is and $v' \neq t$, increment i and repeat.

First, notice that if there is a path from s to t, there will be some execution path (choice of v's) that verifies that s connects to t. Hence, the algorithm is correct. Also notice that the only values we need to keep track of on the work tape are the current vertex, the guessed vertex, and the clock. Each of these can be written in $c \cdot \log n$ space because we can express or encode each vertex in a binary sequence that represents a number up to n and the clock only needs to count up n (a binary counter works fine).

Now that we have shown that s-t connectivity is in NL, we use the idea above to show that any computation in NL is really just s-t connectivity. Define a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ so that each vertex in $\mathcal{G}$ represents a configuration or snapshot of the machine. In order to do this, we need to represent the input tape, input tape head position, current state, finite control, work tape, and work tape head position in some encoding. It takes $n + \log n + c_1 + c_2 + \log n + \log\log n$ bits, respectively (c_1, c_2 are constants) to do this, but if we think about a specific problem with a given fixed (read-only) input x, we need only remember the input tape head position, current state, work tape, and work tape head position, for $\log n + c_1 + \log n + \log\log n$ (logarithmically many) bits. Then given $N(x)$, the problem with input x, we can construct a graph $\mathcal{G}_{N(x)}$ where each vertex is an encoding of the configuration and a directed edge $(u \to v)$ exists iff the configuration represented by v follows from the one represented by u. Then, the problem of deciding whether $x \in L$ for $L \in \mathsf{NL}$ can be

reduced to the question of whether there exists a path from some starting configuration/vertex s to an accepting configuration/vertex t on the graph $\mathcal{G}_{N_L(x)}$.

2.3 Other Complexity Classes

So we can, in effect, forget about NL and just remember s-t connectivity. It is easily apparent, then, that $\mathsf{NL} \subseteq \mathsf{P}$ because we can simply do a depth-first search on any NL problem. Just like NP, there is a corresponding deterministic class for NL: *deterministic log space* (L) is the class of languages that can be determined in logarithmic time. Obviously, $\mathsf{L} \subseteq \mathsf{NL}$. One interesting problem to consider is to construct a restricted s-t connectivity problem which captures L.

Finally, we have the class PSPACE which contains all languages which can be determined by a deterministic TM in polynomial space. We note that the space of all languages that can be determined by a non-deterministic TM in polynomial space is exactly the same as PSPACE (thus $\mathsf{PSPACE} = \mathsf{NPSPACE}$). Specifically, $\mathsf{NSPACE}[f] \subseteq \mathsf{DSPACE}[f^2]$. We know that $\mathsf{L} \neq \mathsf{PSPACE}$ by the Space-Hierarchy Theorem, but that's about all we know of the separation between the containment of complexity classes:

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$$

3 NP-Completeness

The theory of NP-Completeness owes a lot to the reduction-oriented aspect of recursion theory, where problems are shown to be undecidable by the Halting Problem to them. It makes things easier if we can consider a whole bunch of problems that are related in such that a way that we can convert one problem to another and apply similar techniques or results.

3.1 Halting Problem

The Halting Problem, determining whether or not a machine will halt given a specific input, is undecidable: there is no Turing machine which will take, as input, a pair $\langle M, w \rangle$, and will always halt and tell us whether M halts or runs forever on input w. This is one of the major results in computer science and defies the intuition that you should be able to compute anything given enough time. Another problem that is undecidable is whether a recursively enumerable language L is finite or infinite. It would be nice if we could simply say that this problem is similar to the Halting Problem and know that it was undecidable. While it's not quite that easy, one of the main concepts of recursion theory is that if we can reduce the Halting Problem to some seemingly different problem, we show that this different problem is undecidable without resorting to a long, tedious proof to justify our conclusion, in a sense showing the essence of its undecidability.

For our example of checking whether or not a language is finite, first notice that the input for the Halting Problem is a machine M and a string w. We wish to know whether M halts given input w. So construct a machine M_w that will "translate" the problem of halting into the problem of determining whether a language is finite or not. To do this, this machine will ignore its input, instead running as if it were input w. In effect, the machine M_w will simulate M on input w. Notice, however, that $L(M_w)$ is only finite if it is the empty set since we either accept on all inputs (since all inputs are treated as if they are w) or reject on all inputs. Thus, M_w halts iff $L(M_w)$

is infinite. So if we can determine whether a language is finite, we can solve the halting problem. However, since we know that Halting Problem is undecidable, determining whether a language is finite is also undecidable.

In some sense, all you need to know about undecidable problems is the Halting Problem. In a similar vein, we will see in the next section that there are some problems, the NP-Complete problems, which completely characterize NP and so are believed to be intractable (though still decidable, of course). So, if we reduce an NP-Complete problem to some other problem, we will show that this other problem is also intractable in the same way.

3.2 NP-Complete Problems and Reductions

We wish to employ a similar idea as that used above for certain problems in NP. Thus, we would like to have a mechanism to reduce a language L_1 to another language L_2 in a reasonable amount of time.

Definition. A language L_1 *reduces* (in a many-one reduction) to another language L_2 in polynomial time ($L_1 \leq_m^{\mathsf{P}} L_2$) if

1. $x \in L_1 \iff f(x) \in L_2$, and
2. f is deterministic polynomial time computable.

This means that if we wish to reduce Satisfiability (SAT) to Vertex-Cover (VC), we cannot simply wait until we know whether a given formula is satisfiable and then pick a corresponding graph for vertex cover. Such a procedure would not be computable in polynomial time, violating the second condition of the definition. Our goal for reduction is to take a given formula and produce a vertex cover such that when we find out whether the formula is satisfiable, we will know whether the vertex cover exists. In other words, we simply want to encode the formula as a graph. If we can find a function f that accomplishes this, we can say that SAT $\leq_m^{\mathsf{P}}$ VC via f. This is an *expressibility* issue, not a computability issue.

The power of NP-Completeness is that if we have an algorithm for say VC, we automatically have an algorithm for SAT. SAT is connected to thousands of NP-Complete problems, and if we can do one in polynomial time, we can do all of them in polynomial time. There is evidence (since there are thousands of NP-Complete problems and no poly-time algorithm has been found for any of them) that there is no such algorithm. It can be shown that SAT is NP-Complete by using a NTM (non-deterministic Turing Machine), and this is outlined in the text. In effect, we use boolean logic to express what computation is.

One important reduction is SAT $\leq_m^{\mathsf{P}}$ 3-SAT, which puts the variable length clauses of SAT into a more constrained structure. 3-SAT is the language that consists of all satisfiable boolean expressions of the form

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{3} \widehat{x}_{i_j}$$

where $\widehat{x}$ is either x or $\overline{x}$. Thus, $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \overline{x}_5)$ is example of a 3-SAT expression. Note that any 3-SAT expression is also a valid SAT expression, so the reduction from 3-SAT to SAT

is essentially the identity. On the other hand, 2-SAT, which is defined in a similar manner but j ranges only to 2, is not NP-complete; it is decidable in polynomial time as it can be formulated as a graph of logical implications and one can find if the implications are consistent or not in polynomial time.

In order to transform SAT into 3-SAT, we use a construction termed the "butterfly construction". The idea is that for any clause that is longer than three components, we use two components and represent the rest of the clause by a stand-in variable whose inverse is added to the list of components. Consider the following formula:

$$\varphi = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$$

We simply split this formula into 3-clauses as follows:

$$\begin{aligned}\varphi &= (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \\ &= (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee x_5 \vee x_6) \\ &= (x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge (\overline{y}_2 \vee x_4 \vee y_3) \wedge (\overline{y}_3 \vee x_5 \vee x_6)\end{aligned}$$

Notice that this ensures there must be some x_i that is true because if we simply let each y_j be true, the last clause will never be satisfied. Also, if we set one of the x_i to be true, the effect of this propagates to satisfy the whole clause. In other words, once we have one of the literals is true, we can set the y_j to take advantage of this. It looks something like this where $\uparrow$ represents true and $\downarrow$ represents false for the whole literal (including the not symbol):

y_1	$\overline{y}_1$	x_3	y_2	$\overline{y}_2$	y_3	$\overline{y}_3$
$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$

This is the motivation for calling it the butterfly construction.

With the SAT $\leq_m^{\mathsf{P}}$ 3-SAT reduction, we can show many other problems like VC are NP-Complete by reducing 3-SAT to them. In the next lecture, we will examine more such reductions.