ECS 120 Lesson 27 – Intractability, Hierarchy Theorems

Oliver Kreylos

Monday, June 4th, 2001

In our discussion of time complexity and the classes $\mathcal{P}$ and $\mathcal{NP}$, we isolated the $\mathcal{NP}$-complete ones as the hardest problems in $\mathcal{NP}$. We also said that, for practical purposes, $\mathcal{NP}$-complete problems can be treated as *intractable* – as problems where all but the smallest input sizes require a prohibitive amount of running-time. But, since the question "$\mathcal{P} = \mathcal{NP}$?" is still undecided, there might be efficient, polynomial algorithms for $\mathcal{NP}$-complete problems after all. Today we are going to develop the tools to prove one practical problem to be intractable, by showing that any algorithm solving it needs exponential time.

1 Hierarchy Theorems

Intuition tells us that, if a Turing Machine is given more time or space to compute, it should be able to decide more languages. The next two theorems show that this intuition is true – up to some technicalities. Since it is simpler, we start by looking at how an increase of memory increases the amount of problems that can be solved.

1.1 Space Hierarchies

To start, we need to define by how much we have to increase a Turing Machine's available memory in order to make a difference. Some very small increases might not be useful to a machine.

Definition 1 (Space-constructible Functions) *A function $f\colon \mathbf{N} \to \mathbf{N}$ is called* space-constructible*, if there exists a Turing Machine M that, on input $\mathtt{1}^n$, writes $f(n)$ in binary onto its tape, and uses only $O\big(f(n)\big)$ space.*

Space constructible functions are useful in the following way: If a Turing Machine is supposed to be running in $O\big(f(n)\big)$ space, where $f(n)$ is space-constructible, it can scan its input to determine its length n, and can compute the amount of space $f(n)$ it has available to do its computation. If $f(n)$ were not space-constructible, just checking the length of the input and finding out how much space is left would already make the machine exceed its space limit – functions like that are hardly useful.

Luckily, every "reasonable" function is also space-constructible. For example, $f(n) = n^2$ is space constructible: Given input $\mathtt{1}^n$, a machine could count the number of ones using binary notation, using only $\log n$ additional space, and could then calculate n^2 by multiplying the binary notation of n with itself. The standard written multiplication algorithm needs only $O(k)$ space to multiply two k-bit numbers; therefore the additional space needed is another $O(\log n)$. In total, $\mathtt{1}^n$ can be converted to n^2 in binary using only $O(n + \log n) = O(n) = O(n^2)$ space.

Using space-constructible functions, we can state the following hierarchy theorem:

Theorem 1 (Space Hierarchy Theorem) *Let $f\colon \mathbf{N} \to \mathbf{N}$ be a space-constructible function. Then there exists a language A that can be decided in space $O\big(f(n)\big)$, but not in space $o\big(f(n)\big)$.*

The proof of this theorem relies on diagonalization: We construct a Turing Machine B that decides language A in $O\big(f(n)\big)$ space in such a way that it must contradict every Turing Machine that needs less space to run on some input word. We construct B to accept only descriptions of other Turing Machines M, and contradict the machine M given to it on input word $\langle M \rangle$ – the encoding of M itself. We have already seen the "trick" of feeding machines their own descriptions in our diagonalization proof to show the acceptance problem for Turing Machines undecidable.

Using this trick, the constructed language A must be different from every language that can be decided in $o\big(f(n)\big)$ space. The language so constructed is highly abstract – it is not even important to understand which words it contains – and it is not useful for anything but showing the existence of such

a language. Later, we will see how to use mapping reductions to relate other, interesting languages to the "helper language" A.

Casting a lot of details and technicalities aside, here is the Turing Machine B that decides A:

Algorithm On input $\langle M \rangle$,

1. Count the length of the input word $n = |\langle M \rangle|$, compute $f(n)$, and mark off $f(n)$ cells on the tape.
2. Simulate machine M on input word $\langle M \rangle$.
3. If M ever leaves the marked portion of the tape, i. e., uses more than $f(n)$ space, reject.
4. If M loops, reject.
5. If M halts without exceeding its space limit, negate its answer: If M accepts, reject; otherwise, accept.

This algorithm runs in space $O\big(f(n)\big)$: Since $f(n)$ is space constructible, step 1 can be done in $O\big(f(n)\big)$ space. The simulation of M involves storing its configurations, which can at most be $O\big(f(n)\big)$ long. The algorithm can detect if M goes into a loop, because M only uses a finite amount of tape. Thus, its total number of configurations is finite – limited by $2^{O(f(n))}$ – and M must loop if it did not halt after $2^{O(f(n))}$ steps. By counting off steps, B can check this. The counter needs at most $O\big(f(n)\big)$ bits, so it fits into the space allowed for B.

Let us now assume that B' is a machine that also decides language A, but uses less space. Then, when $\langle B' \rangle$ is given to B as input, it will run to completion, and B will contradict B'. Therefore, their languages cannot be identical – they disagree on word $\langle B' \rangle$. This shows that no Turing Machine can decide A in $o\big(f(n)\big)$ space.

The space hierarchy theorem provides us with a hierarchy of proper inclusions of classes $\mathrm{SPACE}\big(f(n)\big)$ into each other:

- $\mathrm{SPACE}(n) \underset{\neq}{\subset} \mathrm{SPACE}(n^2)$.
- $\mathrm{SPACE}(n^{k_1}) \underset{\neq}{\subset} \mathrm{SPACE}(n^{k_2})$, where $k_1 < k_2$.

- $\mathrm{SPACE}(n^k) \underset{\neq}{\subset} \mathrm{SPACE}(2^n)$, for any $k \geq 0$.
- $\mathrm{PSPACE} \underset{\neq}{\subset} \mathrm{EXPSPACE}$.

1.2 Time Hierarchies

The same reasoning about space complexity hierarchies can be applied to time complexity as well. Again, we need a definition of "reasonable" functions:

Definition 2 (Time-constructible Functions) *A function $f\colon \mathbf{N} \to \mathbf{N}$ is called* time-constructible*, if there exists a Turing Machine M that, on input $\mathtt{1}^n$, writes $f(n)$ in binary onto its tape, and uses only $O\big(f(n)\big)$ time.*

Again, this is no real restriction, because virtually every function that we are interested in is time-constructible. Using these functions, we can state a theorem about time hierarchies similar to the one about space hierarchies, with one little but crucial difference. We already observed that space is more powerful than time (at least for Turing Machines), because space can be re-used, whereas time cannot. The effect of this is, that the time hierarchy theorem does not make as sharp a distinction between complexity classes as its space equivalent.

Theorem 2 (Time Hierarchy Theorem) *Let $f\colon \mathbf{N} \to \mathbf{N}$ be a time-constructible function. Then there exists a language A that can be decided in time $O\big(f(n)\big)$, but not in time $o\big(f(n)/\log f(n)\big)$.*

The only difference between these two theorems – apart from replacing space by time – is the factor $\log f(n)$. It stems from the fact that the simulating Turing Machine in the proof for this theorem must perform additional work to keep track of the configuration of the machine it simulates. This additional work does not use additional space, but additional time on the order of $\log f(n)$. This restricts us to broader distinctions between time complexity classes. Using this theorem, we would *not* be able to show that there is a language that can be decided in $O(n \log n)$ but not in $O(n)$. Luckily, we are currently not interested in these fine distinctions anyway; the theorem is powerful enough to distinguish between the classes $\mathrm{TIME}(n^k)$ and $\mathrm{TIME}(2^n)$.

Apart from the obvious difference, the proof works in the same way as for space hierarchies: We construct a machine B that, by diagonalization, can

not have the same language as any other machine that runs in less time – less by a factor of $\log f(n)$, that is. Here is the basic algorithm for B, again leaving out many technical details:

Algorithm On input $\langle M \rangle$,

1. Count the length of the input word, $n = |\langle M \rangle|$, and compute $f(n)$.
2. Simulate machine M on input $\langle M \rangle$.
3. If M does not halt in $f(n)/\log f(n)$ steps, reject.
4. If M halts in the time allotted, negate its answer: If M accepts, reject; otherwise, accept.

Since $f(n)$ is time-constructible, the first step can be performed in $O\big(f(n)\big)$ time. For each step of the simulation of M, however, B has to decrease the counter of steps by one. Since this counter starts as $f(n)/\log f(n)$, its binary encoding is $O\big(\log f(n)\big)$ bits long, and decrementing it takes an additional $O\big(\log f(n)\big)$ steps for each simulated step of M. This means, the total time complexity for simulating a machine M of time complexity $f(n)/\log f(n)$ is $O\big(f(n)\big)$, as claimed. The same diagonalization argument as for the space hierarchy proof works here; there can be no machine that accepts the same language in time $o\big(f(n)/\log f(n)\big)$.

As a result from the time hierarchy theorem, we have proper nestings of the following complexity classes, amongst others:

- $\mathrm{TIME}(n) \underset{\neq}{\subset} \mathrm{TIME}(n^2)$.
- $\mathrm{TIME}(n^{k_1}) \underset{\neq}{\subset} \mathrm{TIME}(n^{k_2})$, where $k_1 < k_2$.
- $\mathrm{TIME}(n^k) \underset{\neq}{\subset} \mathrm{TIME}(2^n)$, for any $k \geq 0$.
- $\mathcal{P} \underset{\neq}{\subset} \mathrm{EXPTIME}$.

2 An EXPSPACE-complete Problem

We are now going to use the tools provided by space- and time-hierarchies to show that there is an interesting problem that can not be solved in polynomial time. To introduce this problem, we have to add a new operator to the definition of regular expressions.

2.1 Regular Expressions with Exponentiation

Regular expressions over an alphabet Σ, $\mathcal{R}(\Sigma)$, are a convenient way to specify regular languages over the same alphabet. In our old definition, regular expressions were built by combining three base cases into more complex expressions by using the three regular operations union, concatenation and Kleene Star. Regular expressions R, and the languages $L(R)$ described by them, are defined as follows:

1. $\emptyset \in \mathcal{R}(\Sigma)$ is a regular expression over alphabet Σ. Its language is $L(\emptyset) = \emptyset$, the empty language.
2. $\epsilon \in \mathcal{R}(\Sigma)$. Its language is $L(\epsilon) = \{\epsilon\}$, the language consisting only of the empty word.
3. For any $a \in \Sigma$, $a \in \mathcal{R}(\Sigma)$, and $L(a) = \{a\}$.
4. If $R_1, R_2 \in \mathcal{R}(\Sigma)$, then $(R_1 \cup R_2) \in \mathcal{R}(\Sigma)$, and $L\big((R_1 \cup R_2)\big) = L(R_1) \cup L(R_2)$ is the union of $L(R_1)$ and $L(R_2)$.
5. If $R_1, R_2 \in \mathcal{R}(\Sigma)$, then $(R_1 \circ R_2) \in \mathcal{R}(\Sigma)$, and $L\big((R_1 \circ R_2)\big) = L(R_1)L(R_2)$ is the concatenation of $L(R_1)$ and $L(R_2)$.
6. If $R \in \mathcal{R}(\Sigma)$, then $(R^*) \in (\Sigma)$, and $L\big((R^*)\big) = L(R)^*$ is the Kleene Star of $L(R)$.

To allow expressing some languages in a more convenient way, we will now add another operator, *exponentiation*, to the regular operators:

7. If $R \in \mathcal{R}(\Sigma)$ and $n \in \mathbf{N}_0$ is a non-negative integer, then $(R \uparrow n) \in \mathcal{R}(\Sigma)$, and $L\big((R \uparrow n)\big) = L(R)^n = \underbrace{L(R)L(R)\ldots L(R)}_{n \text{ times}}$ is the n-times concatenation of $L(R)$ with itself.

Having the exponentiation operator does not change the power of regular expressions; they still generate the same class of languages. After all, each exponentiation $(R \uparrow n)$ can be replaced by replicating expression R n times.

2.2 The Equality Problem for Regular Expressions with Exponentation

One important question when given two regular expressions with exponentation $R_1, R_2 \in \mathcal{R}(\Sigma)$ over the same alphabet Σ is to find out whether the two generate the same language, i. e., whether $L(R_1) = L(R_2)$. Posed as a language, this problem becomes

$$\mathrm{EQ}_{\mathrm{REGEXP}} = \left\{ \langle R_1, R_2 \rangle \;\middle|\; R_1, R_2 \in \mathcal{R}(\Sigma) \wedge L(R_1) = L(R_2) \right\}$$

We already know that this problem is decidable: We can convert the two regular expressions to NFAs, those to DFAs, minimize both DFAs and compare the structure of the resulting minimal automata. If the structures are equal, the languages must be identical. But, as it turns out, when allowing exponentiation, the algorithm to check for equality becomes exponential in time and space complexity.

The algorithm to check for equivalence we have seen for normal regular expressions almost works for regular expressions with exponentiation as well: The only additional step is to remove all exponentations $(R \uparrow n)$ by explicitly replicating the affected subexpressions R n times. But this step can increase the size of the expression exponentially: If the exponent n is encoded in binary, the number of repetitions we have to perform is exponential in the length of n, and, in the worst case, exponential in the total length of the input expression. A simple example where this happens is the expression $(\mathtt{a} \uparrow 1111111111)$; it is only 14 characters long, but the expanded version, $\mathtt{aaa\ldots a}$ is 1023 characters long (1023 is 1111111111 in binary).

This space-exponential algorithm does not mean that there can be no better one, but as it turns out, $\mathrm{EQ}_{\mathrm{REGEXP}}$ is EXPSPACE-complete. This completeness is defined similarly to $\mathcal{NP}$-completeness:

Definition 3 (EXPSPACE-completeness) *A language A is EXPSPACE-complete, if and only if*

1. $A \in$ EXPSPACE*, and*

2. *for any* $B \in \text{EXPSPACE} : B \leq_P A$*, i. e., every other problem in EXPSPACE is polynomial-time reducible to* A*.*

To prove that $\text{EQ}_{\text{REGEXP}}$ is EXPSPACE-complete, we have to show a way to reduce every EXPSPACE problem to it. We will use a computation chain reduction similar to the ones we used to show that PCP and ALL_{CFG} are undecidable.

Altogether, by showing that $\text{EQ}_{\text{REGEXP}}$ is EXPSPACE-complete, we know that if there exists any problem that requires exponential space to run, $\text{EQ}_{\text{REGEXP}}$ must also require exponential space and can have no better algorithm. But from the hierarchy theorems we saw earlier we know that there is a language in $\text{EXPSPACE} \setminus \text{PSPACE}$ – though a highly abstract one – and that $\text{EQ}_{\text{REGEXP}} \in \text{EXPSPACE} \setminus \text{PSPACE}$ as well. Now, if a problem requires exponential space, it must also require exponential time: A Turing Machine can only write a single character per computation step, so if it has to write an exponential amount of characters, it must at least use an exponential number of steps. Then $\text{EQ}_{\text{REGEXP}}$ must be an element of $\text{EXPTIME} \setminus \mathcal{P}$. Therefore we have seen the first example of a problem of practical importance that is provably intractable, even if the classes $\mathcal{P}$ and $\mathcal{NP}$ should turn out to be equal.