CSCC63 – Week 9

This week: Let's finish off our discussion of the class $\mathcal{NP}$ by looking at Cook's theorem, and then start our discussion of space complexity.

The Cook-Levin Theorem

The Cook-Levin theorem will prove the assumption we've been making up to this point: that 3SAT is $\mathcal{NP}$-complete.

When we've proven that other problems L are $\mathcal{NP}$-complete, we've done so by assuming that 3SAT is $\mathcal{NP}$-complete, and then showing that 3SAT $\leqslant_p L$. This is because it is easier to give one reduction than to show that $\forall A \in \mathcal{NP}$, $A \leqslant_p L$.

But how can we possibly prove this?

A Natural $\mathcal{NP}$-complete Problem

Idea: We don't actually need to reduce from every possible A in $\mathcal{NP}$ – there's one language that "naturally" solves all of them!

The following problem is $\mathcal{NP}$-complete:

$$\mathrm{A}_{\text{B-NTM}} = \left\{ \langle M, w, \#^r \rangle \;\middle|\; \begin{array}{l} M \text{ is a verifier that accepts some certificate } c \\ \text{of length at most } |w|^k \text{ for } w \text{ within } r \text{ steps} \end{array} \right\}$$

This problem is in $\mathcal{NP}$: the certificate is an accepting computation path of length $\leq r$.

To see why this has to be $\mathcal{NP}$-hard, consider the example:

- HAM-PATH $\in \mathcal{NP}$, so we can build an verifier M that decides whether a path P for $\langle G, s, t \rangle$ is a certificate for HAM-PATH using $\mathcal{O}(n^{k'})$ steps for some $k' \in \mathbb{N}$.
- We can set $w = \langle G, s, t \rangle$ and r to be the worst-case time M can take for an input of size $|w|$ (00for any certificate).
- If we do so, then $\langle M, w, \#^r \rangle \in \mathrm{A}_{\text{B-NTM}}$ iff $\langle G, s, t \rangle \in$ HAM-PATH.

A similar argument will clearly work for any $A \in \mathcal{NP}$.

So we really only need to show that $\mathrm{A}_{\text{B-NTM}} \leqslant_p$ 3SAT – that is, we need to find a way to embed a TM operation into a boolean formula.

This is going to be a bit like our SUBSET-SUM proof, in that we're going to build a ϕ that emulates the important behaviour of our original problem (it also involves a huge table). But this time we're starting with a TM, not a 3CNF formula.

This will also look a lot like our proof for the PCP*, since we're using something other than a TM to emulate a TM. In fact, we can easily show that a bounded version of the* PCP *is $\mathcal{NP}$-complete by adapting the proof from a couple from weeks ago.*

Building a Boolean Formula from a TM

So we'd like to build a boolean formula ϕ that encodes the operation of a TM M.

Let's start by looking at a deterministic TM, and then we can generalize to the deterministic TM that also takes a certificate.

If M is deterministic, we can describe its operations by listing its configurations: e.g.,

$$C_0 = q_0 w_0 w_1 \dots w_{n-1}, C_1 = t_{0,1} q_1 w_1 \dots w_{n-1}, \dots .$$

We can also stack its configurations like so:

$$\begin{aligned} C_0 = &\quad q_0 w_0 w_1 \dots w_{n-1} \\ C_1 = &\quad t_{0,1} q_1 w_1 \dots w_{n-1} \\ &\quad \vdots \end{aligned}$$

We'll turn this stack into a table. Firstly, we note that there is some $k \in \mathbb{N}$ such that M cannot take more that n^k steps, since we're considering an M that halts in polynomial time (n^k is the r from the previous page).

We also note that, since M cannot move its head more than one square per step, it cannot look at more than n^k memory spaces. So we can buffer our stack with $\sqcup$ characters like so:

TM operation:

$\#$	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	$\#$
$\#$	$t_{1,0}$	q_1	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	$\#$
$\#$										$\#$
$\#$										$\#$
$\vdots$	$\vdots$	$\vdots$			$\vdots$					$\vdots$
$\#$	$t_{n^k-1,0}$	$t_{n^k-1,1}$			$\dots$				t_{n^k-1,n^k-1}	$\#$

Notice that we've placed $\#$ symbols on the ends of the tape. These just act as place markers for us.

So we have an $n^k \times (n^k + 2)$ table that completely describes the operation of M on w. We'll have to build a ϕ that encodes this M.

> **Question**: *What are the properties of the table we need to encode into ϕ if we want it to emulate the TM operation?*

There are four major properties we care about:

1. Every cell contains exactly one character.
2. The characters on the first row match C_0 (it contains q_0 and the input).
3. There is a q_{accept} character somewhere on the table.
4. Every row in the table follows from the one above given the operation of M.

There are other properties we might want to encode, e.g., there is only one head character per row. But we can derive these properties from the four listed (e.g., from properties 2. and 4.).

So if we can build a ϕ encoding these four properties, we're done! (for deterministic TMs)

Since ϕ is already a conjunction of lots of smaller clauses, we can compose it of four different clause sets:

- ϕ_{cell} is a boolean formula that is true iff property 1. holds.
- ϕ_{start} is a boolean formula that is true iff property 2. holds.
- ϕ_{accept} is a boolean formula that is true iff property 3. holds.
- ϕ_{move} is a boolean formula that is true iff property 4. holds.

If we can build these formulas, then $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$.

To start building these formulas, let's fix the variables we use: We say the variable $x_{i,j,s}$ is true if cell $[i, j]$ in the above table holds the character s.

Encoding ϕ_{cell}:

To encode this property, we need to ensure that every cell contains at least one character that is true, but not more than one.

To ensure that at least one character is in cell [i, j], we need to state that one of the character variables in that cell is true. So we write

$$\bigvee_{s} x_{i,j,s}.$$

To ensure that only one character is in the cell, we state that for any two characters s and s', at least one of their character variables is false:

$$\bigwedge_{s,s',s\neq s'} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,s'}}\right).$$

We can see that the second formula is already in 2CNF, while the first is a disjunction of many variables.

We can turn a 2CNF clause into a 3CNF clause by repeating one of the literals (or adding a dummy literal that we force elsewhere to be false).

We can turn a disjunction of many variables into a 3CNF clause like so:

We start with a clause such as $a \vee b \vee c \vee d$, and we separate it by adding new variables to get $(a \vee b \vee y_1) \wedge (\overline{y_1} \vee c \vee d)$.

So we can encode property 1. for a single cell at [i, j] into a 3CNF formula. If we call this formula $\phi_{cell:i,j}$, then clearly the ϕ_{cell} we want is just

$$\phi_{cell} = \bigwedge_{i,j} \phi_{cell:i,j}.$$

So we have ϕ_{cell}.

Encoding ϕ_{start}:

To encode ϕ_{start}, we just manually set the characters on the first row to be C_0:

$$x_{0,0,\#} \wedge x_{0,1,q_0} \wedge x_{0,2,w_0} \wedge \ldots \wedge x_{0,n^k,\sqcup} \wedge x_{0,n^k+1,\#}$$

Every variable above is already in a 1CNF clause, so we can turn it into a 3CNF clause using the same trick as we used above.

So we have ϕ_{start}.

Encoding ϕ_{accept}:

To encode ϕ_{accept}, we need to specify that the q_{accept} character can be found somewhere on the table. So we just write that one of its variables is true:

$$\bigvee_{i,j} x_{i,j,q_{accept}}.$$

We can turn this into a 3CNF using the trick described in the ϕ_{cell} section.

So we have ϕ_{accept}.

Encoding ϕ_{move}:

This is the hard one. The trick is to remember (from assignment 1) that two neighbouring configurations can differ by only three characters. So if we look at any two adjacent rows in the table, we need to only worry about windows three characters long, e.g., in the table above,

#	q_0	w_0
#	$t_{1,1}$	q_1

q_0	w_0	w_1
$t_{1,1}$	q_1	w_1

$\sqcup$	$\sqcup$	$\sqcup$
$\sqcup$	$\sqcup$	$\sqcup$

Now, there may be many possible allowable windows of this sort – they just need to be consistent with the TM operation. So, for example, the

q_0	w_0	w_1
$t_{1,1}$	q_1	w_1

and

$\sqcup$	$\sqcup$	$\sqcup$
$\sqcup$	$\sqcup$	$\sqcup$

might occur in many places in the table. We say a window is legal if, like the windows above, it is consistent with the TM operation given the local knowledge we have in the window. So, for example, a window like

q_0	w_0	w_1
$t_{1,1}$	q_1	q_1

would not be legal, not just because it doesn't occur, but because it can't – the TM can't have two heads at once.

In order to ensure that the steps of the TM are encoded into ϕ, ϕ_{move} will have to check that every window is legal:

$$\phi_{move} = \bigwedge_{i,j} \big(\text{The (i, j) window is legal}\big).$$

Now, suppose that we write a window A at $[i, j]$ as

$A_{i,j}$	$A_{i,j+1}$	$A_{i,j+2}$
$A_{i+1,j}$	$A_{i+1,j+1}$	$A_{i+1,j+2}$

.

Then the property that the (i, j) window is legal can be encoded as:

$$\bigvee_{A:A \text{ is legal}} \left(x_{i,j,A_{i,j}} \wedge x_{i,j+1,A_{i,j+1}} \wedge x_{i,j+2,A_{i,j+2}} \wedge x_{i+1,j,A_{i+1,j}} \wedge x_{i+1,j+1,A_{i+1,j+1}} \wedge x_{i+1,j+2,A_{i+1,j+2}}\right).$$

We note that because the A are just the 2×3 windows that could be consistent with the operation of M, the number of A is finite, and so the above formula exists – and is even bounded in size for all i and j.

We leave it as an exercise to show that this formula can be written in 3CNF, but we note that when we do, the size of the formula depends only on M, not on the input size. So we can get a 3CNF formula $\phi_{move:i,j}$.

Given $\phi_{move:i,j}$, we can just write $\phi_{move} = \bigwedge_{i,j} \phi_{move:i,j}$. So we have ϕ_{move}.

Putting it all Together

Since we've defined ϕ_{cell}, ϕ_{start}, ϕ_{accept}, and ϕ_{move}, we also have $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, as we stated above.

The only problem that remains is to ensure that the reduction is polynomial in size.

But we note that ϕ_{cell} is just the same formula for all cells. It is $\mathcal{O}(n^{2k})$ in size. The same holds for ϕ_{accept} and ϕ_{move} (note that the individual $\phi_{move:i,j}$ may be very large, but it depends on M, not on the size n of w).

We can also see that ϕ_{start} has a clause for every cell in the first row of the table: it has size $\mathcal{O}(n^k)$.

So our entire reduction is of size $\mathcal{O}(n^{2k})$, and so the reduction is polynomial in time.

To finish off, we remember that we were assuming that M was deterministic. To make it non-deterministic:

- We add legal windows when defining ϕ_{move} to cover the possibility of non-determinism during the program run.

  Basically, nothing in our construction required the TM to be deterministic – it just made the configuration table easier to viualize. But the same idea still works when the TM is non-deterministic.

- For future reference, though, we'll ue a very particular type of NTM in this construction: if a language L is in $\mathcal{NP}$, then it has a polytime verifier V. Our NTM for L will be of the form:

```
Let $N$ = "On input $x$:
  1. Non-deterministically choose a certificate $c$ for $x$.
  2. Run $V$ on $\langle x, c\rangle$.
  3. If it accepts, accept, otherwise reject.
```

  We oberve that the certificate choice is the only non-determinism in N.

Either way, our reduction gives us a ϕ in polynomial time, and so $\mathrm{A}_{\text{B-NTM}} \leqslant_p \mathrm{3SAT}$.

So 3SAT is $\mathcal{NP}$-complete, as desired.

And that finishes our section on the class $\mathcal{NP}$. Let's move on to space complexity.

Space Complexity

Now, the class $\mathcal{NP}$ decribes the worst-case time complexity of the NTMs that we can use to solve a problem. But there are other ways of looking at complexity. In fact, any time there's a constrained resource that we need for a calculation we can reasonably ask how much of that resource we'll need:

- We can ask how much memory we'll need – we might, for example, be in a situation where we have a server farm somewhere that we can devote to solving some problem that we're interested in. The calculation may take a few months, but if we don't need the answer today that might be OK for us. But we'll want to know how many resources we need to allocate for this problem. In fact, we'll need to be sure that our server farm is big enough.

- We sometimes run randomized algorithms and so need source of random bits. These bits are expensive, so we can resonably ask how many we'll need in order to solve our problem.

  We're not covering probability right at the moment, though, so we'll leave that discussion for another time.

The space complexity of a problem or program addresses the first of these two constraints. We'll be trying to measure the amount of memory needed to solve a problem. In terms of TMs, this is the number of tape cells used.

But wait – if a problem needs lots of time, wouldn't it need lots of space as well? Is space complexity even different from the time complexity of a program?

Well, we don't actually know: we strongly believe that the worst-case time complexity of a program can differ wildly from its space complexity, but we haven't proven it yet.

To see why this question might be different from the time complexity we've already seen, recall that we strongly suspect that 3SAT cannot be solved in a polynomil mount of time.

But consider the following program:

```
Let **3SAT-SOLVER** = "On input $\langle \phi \rangle$:
  1. Check that $\phi$ is a 3CNF and reject if it is not.
  2. Initialize a $k$-bit counter $i$, where $k$ is the number of variable in $\phi$.
  3. For $i = 0$ to $2^k - 1$:
  4.     Read a truth assignment $\tau_i$ from the binary representation of $i$.
     We turn the $0$s in the counter to $F$s and the $1$s to $T$s.
     E.g., if $k = 4$ and $i = 0000$ , then $\tau_i = FFFF$,
         and if $i = 0100$ then $\tau_i = FTFF$.
  5.     Check if $\tau_i$ is a satisfying assignment for $\phi$.
  6.     If it is, accept.
  7. Reject.
```

Now, we can see that this program takes only the memepry needed for the binary counter, plus the space needed to check the truth assignment τ_i. Clearly the space needed for the counter is polynomial in the size of ϕ, so the only question is, how much space do we need to check the τ_i?

What we're going to argue is that we can check the τ_i in polynomial time, and so we only need a polynomial amount of space for it as well. In general, the space required to use for a problem is no more than the time required.

We've already touched on this idea: If a TM takes at most k steps in its operation, then it also uses at most k memory cells – it can only look at one cell per step.

Remember question 4 from assignment 1!

But a TM can also look at a single cell more than once. You can see that we reuse the bits in the counter of our above program, for example. *So it is possible to re-use steps in space, but not time.*

Otherwise, the complexity classes based on space are similar to those based on time: if a TM M halts on all inputs of length n using at most $f(n)$ tape cells, it is a program that has a space complexity of $f(n)$.

This extends to definitions for languages:

Definition: SPACE$(f(n)) = \{L \mid L$ can be decided by a $\mathcal{O}(f(n))$-space TM.$\}$

Definition: NSPACE$(f(n)) = \{L \mid L$ can be decided by a $\mathcal{O}(f(n))$-space NTM.$\}$

As before, we'll look at the problems that can be solved or verified in an efficient (polynomial) amount of space:

Definition: $\mathcal{PS} = \bigcup_{k \in \mathbb{N}}$ SPACE(n^k).

We'll refer to the class $\mathcal{PS}$ as "PSPACE".

Definition: $\mathcal{NPS} = \bigcup_{k \in \mathbb{N}}$ NSPACE(n^k).

We'll refer to the class $\mathcal{NPS}$ as "NPSPACE".

We're considering all tape memory used here, including the memory used for the input. So any TM that needs to look at the whole input will have to use at least $\mathcal{O}(n)$ space. Can you see why?

We could also consider a model that uses a read-only input tape and a regular working tape. In this model we only count the memory used on the working tape. This would allow us to define a class of $\mathcal{O}(\log(n))$-space languages, which has a number of interesting properties. For today we'll only worry about the space used by regular TMs, though.

The big question we want to answer is how these complexity classes compare to the ones we already know – if we measure the space used instead of time, do we get any extra power?

Once again, the answer is that we're not sure. But it looks like we do get more power.

Firstly, we have seen that any TM that takes $\mathcal{O}(n^k)$ time also takes $\mathcal{O}(n^k)$ space. So $\mathcal{P} \subseteq \mathcal{PS}$. Similarly, $\mathcal{NP} \subseteq \mathcal{NPS}$.

It is also clear that since any TM is also an NTM, that $\mathcal{PS} \subseteq \mathcal{NPS}$.

Question: *What about $\mathcal{NP}$ and $\mathcal{PS}$?*

We're really already answered this question, though – if you look above to our **3SAT-SOLVER** program, you'll see that the program uses a polynomial amount of space (in fact, our demonstration that this was possible was how we argued that space complexity might be different from time complexity).

We can generalize this idea to any problem in $\mathcal{NP}$.

Recall that we can write the nondeterministic choices made by an NTM as a string – we just state which branch of the computation tree is followed.

Also recall that we can move the choice of this string to the beginning of the program as a certificate. So we can simulate an NTM by running deterministically checking the paths of its computation tree for every possible certificate:

```
On input $x$:
1. For every certificate $C$:
2.   Try to use $C$ to verify $x$.
3.   If $C$ is a valid certificate for $x$, accept.
4. Reject.
```

The number of certificates is generally exponential, so this will generally take an exponential amount of time.

But the certificates are polynomial in size, and so we can loop over all of them in polynomial space (treat the strings as numbers and increment them by one every loop iteration). Since the inner loop requires only polynomially more space, the entire process takes a polynomial amount of space.

So $\mathcal{NP} \subseteq \mathcal{PS}$.

Here's something more surprising: $\mathcal{PS} = \mathcal{NPS}$!

Why? This is a result of something called **Savitch's Theorem.**

Savitch's Theorem:

For any $f : \mathbb{N} \to \mathbb{R}^+$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}([f(n)]^2)$.

Since this blowup is polynomial in size, we the extra space needed to simulate an NTM on a deterministic TM will not be enough to make any problem that is polynomial in the non-deterministic space grow into a super-polynomial problem in deterministic space.

Question: *Why is this possible, and why doesn't it help us determine whether* $\text{P} \neq \text{NP}$*?*

Proof Sketch:

The overall idea here actually starts off a lot like the proof of the Cook-Levin theorem from earlier[1]. We still want to build table of TM configurations - suppose we have an NTM the takes $f(n)$ steps and n^k memory cells:

$\#$	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	$\#$
$\#$	$t_{1,0}$	q_1	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	$\#$
$\#$										$\#$
$\#$										$\#$
$\vdots$	$\vdots$	$\vdots$			$\vdots$					$\vdots$
$\#$	$t_{f(n),0}$	$t_{f(n),1}$		$\dots$					$t_{f(n),n^k-1}$	$\#$

But this time we're listing the TM configurations of an NTM N and trying to generate them using a deterministic TM M.

Now, if we're going to try to store this whole table, and if the time taken by N is exponential in the input size, then we'd an exponential amount of space to do the work – the table would have an exponential number of rows.

And you can see that the **3SAT-SOLVER** *program indeed takes exponential time.*

But there's something else we can do: we can pick a configuration from halfway down the table:

$\#$	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	$\#$
$\#$	$t_{f(n),0}$	$t_{f(n),1}$			$\dots$				$t_{f(n),n^k-1}$	$\#$
$\#$	$t_{f(n)/2,0}$	$t_{f(n)/2,1}$	$t_{f(n)/2,2}$			$\dots$			$t_{f(n)/2,n^k-1}$	$\#$

So in order to look at what happens on configuration $C_{f(n)}$, i.e., on the step $f(n)$ of the program, we're looking at the possible configurations at step $f(n)/2$. Note that we've swapped the order of the configuration here – it'll make the next steps more clear.

To show that we can reach the configuration $C_{f(n)}$ from C_0:

- We loop over all configurations $C_{f(n)}/2$.
- For each of these configurations, we check whether we can reach the configuration $C_{f(n)}/2$ from C_0.
- We also check whether we can reach the configuration $C_{f(n)}$ from $C_{f(n)}/2$.

[1]In fact, Walter Savitch was a student of Stephen Cook, and the two theorems are related. Historically, Savitch's theorem came first, and the Cook-Levin theorem built in the ideas here.

- If we can show both of these things, then we have the proof we want. If not, we move to the next possible $C_{f(n)}/2$.

  If we run out of possible $C_{f(n)}/2$ configurations we reject.

So how do we check whether we can get from C_0 to $C_{f(n)}/2$ in $f(n)/2$ steps?

We check to see whether there's a configuration $C_{f(n)}/4$ at step $f(n)/4$ such that we can reach $C_{f(n)}/4$ from C_0 and $C_{f(n)}/2$ from $C_{f(n)}/4$:

#	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	#
#	$t_{f(n),0}$	$t_{f(n),1}$	$\dots$						$t_{f(n),n^k-1}$	#
#	$t_{f(n)/2,0}$	$t_{f(n)/2,1}$	$t_{f(n)/2,2}$	$\dots$					$t_{f(n)/2,n^k-1}$	#
#	$t_{f(n)/4,0}$	$t_{f(n)/4,1}$	$t_{f(n)/4,2}$	$\dots$					$t_{f(n)/4,n^k-1}$	#

Once we're done with this we'll tro to show that we can get from $C_{f(n)/2}$ to $C(f(n))$ by looking for a configuration at step $3f(n)/4$:

#	q_0	w_0	w_1	$\dots$	w_{n-1}	$\sqcup$	$\sqcup$	$\dots$	$\sqcup$	#
#	$t_{f(n),0}$	$t_{f(n),1}$	$\dots$						$t_{f(n),n^k-1}$	#
#	$t_{f(n)/2,0}$	$t_{f(n)/2,1}$	$t_{f(n)/2,2}$	$\dots$					$t_{f(n)/2,n^k-1}$	#
#	$t_{3f(n)/4,0}$	$t_{3f(n)/4,1}$	$t_{3f(n)/4,2}$	$\dots$					$t_{3f(n)/4,n^k-1}$	#

Note that we re-use the space in the fourth row of the table.

And how do we find the configuration t step $f(n)/4$? We look for the configuration at step $f(n)/8$ *and* $f(n)/16$*,* $f(n)/32$*, …,* $f(n)/2^{\log_2 f(n)}$.

#	q_0	w_0	w_1	$\dots$	w_{n-1}	$\dots$	$\sqcup$	#
#	$t_{f(n),0}$	$t_{f(n),1}$	$\dots$				$t_{f(n),n^k-1}$	#
#	$t_{f(n)/2,0}$	$t_{f(n)/2,1}$	$t_{f(n)/2,2}$	$\dots$			$t_{f(n)/2,n^k-1}$	#
#	$t_{f(n)/4,0}$	$t_{f(n)/4,1}$	$t_{f(n)/4,2}$	$\dots$			$t_{f(n)/4,n^k-1}$	#
#	$t_{f(n)/8,0}$	$t_{f(n)/8,1}$	$t_{f(n)/8,2}$	$\dots$			$t_{f(n)/8,n^k-1}$	#
$\vdots$	$\vdots$	$\vdots$	$\vdots$				$\vdots$	$\vdots$
#	$t_{f(n)/2^{\log_2 f(n)},0}$	$t_{f(n)/2^{\log_2 f(n)},1}$	$t_{f(n)/2^{\log_2 f(n)},2}$	$\dots$			$t_{f(n)/2^{\log_2 f(n)},n^k-1}$	#

Once we've expanded this table enough we'll have two adjacent configurations – we'll have a possible C_1 at the bottom of the table. And we can test whether C_1 follows from C_0 normally.

And as we've said, we can re-use these rows to show, e.g., that $C_{f(n)}$ follows from $C_{f(n)}/2$.

Since at each step we choose a halfway configuration, the number of rows we'll need is logarithmic in $f(n)$. So the question is, how large can $f(n)$ be? How many steps can N possibly take?

But remember that TM that repeats configurtions must loop – so every row of the table must be distinct. So the number of possible rows is bounded by the number of configurations. We have n^k memory cells, and $|Q \cup \Gamma|$ possible cell contents. So the total number of configurations is about $|Q \cup \Gamma|^{n^k}$, and the number of rows is $\log_2\left(|Q \cup \Gamma|^{n^k}\right) = n^k \log_2\left(|Q \cup \Gamma|\right)$. So the number of table cells we'll need is $\mathcal{O}(n^{2k})$, as we have stated.

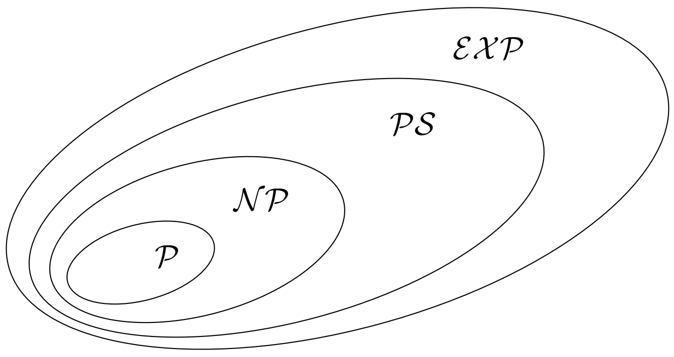
You can see that the actual time for the search is still exponential, though, so the approach doesn't do anything for the $\mathcal{P}$ vs. $\mathcal{NP}$ question.

So here's what we know:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS} = \mathcal{NPS} \subseteq \mathcal{EXP}$$

Note: $\mathcal{EXP}$ is just the exponential time analogue to $\mathcal{P}$.

None of the subset inclusions have yet been proven to be proper, but we know that at least one of them must be. *We strongly suspect that all of the classes are different:*



Reductions in $\mathcal{PS}$:

We can use polytime reductions in relation to $\mathcal{PS}$ in the same way that we do when looking at $\mathcal{P}$ and $\mathcal{NP}$:

- If $A \leqslant_p B$ and $B \in \mathcal{PS}$, then $A \in \mathcal{PS}$.
- If $A \leqslant_p B$ and $A \notin \mathcal{PS}$, then $B \notin \mathcal{PS}$.

The ideas of class hardness and completeness carry over, as well:

A language L is $\mathcal{PS}$-complete if:

- $L \in \mathcal{PS}$.
- $\forall A \in \mathcal{PS}, A \leqslant_p L$.

Question: *Why do we use polytime reductions rather than polyspace reductions?*

We generally use reductions to show that a problem is hard – to show, e.g., that B is $\mathcal{NP}$-complete, and so likely not in $\mathcal{P}$. This only works when the power we have for the reduction is limited.

If our reductions in $\mathcal{PS}$ were allowed to use polynomial space instead of time, we could solve any problem in $\mathcal{PS}$ as part of the reduction. This would make all non-trivial languages in $\mathcal{P}$ $\mathcal{PS}$-complete!

Just like in the class $\mathcal{NP}$, we know a $\mathcal{PS}$-complete language: the True Quantified Boolean Formulas (TQBF):

$$\text{TQBF} = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified boolean formula.}\}$$

A *fully quantified* boolean formula is a ϕ just like the ones in SAT, except every variable has a quantifier: e.g.,

$$\phi = \forall x, y, \exists z, (x \vee z) \wedge (\overline{y} \vee z).$$

The proof that this is $\mathcal{PS}$-complete is somewhat similar to the proof of the Cook-Levin theorem that we saw last week, with a space-saving mechanism a little bit like the one used in the proof of Savitch's theorem.

You can think of an $\mathcal{NP}$ problem like SAT *or* 3SAT *as a problem in which there is only a single existential quantifier "$\exists$" that covers all of the variables. Similarly,* co-$\mathcal{NP}$ *uses a single universal quantifier "$\forall$". This is a generalization.*

Idea: We can treat choices made under quantifiers as being made by adversaries in a game:

- If I say that there is an input x that makes something like a boolean formula ϕ true ("$\exists$"), I'm saying I could tell you that x, so long as I can find it.
- If I say that for all inputs y, ϕ true ("$\forall$"), I'm saying that you can't tell me an input y that makes that ϕ false.
- If I chain many quantifiers together, we can take turns choosing inputs – I choose the ones covered by"$\exists$" and you choose the ones covered by "$\forall$". If I say the ϕ is true with these quantifiers, what I'm saying is that even if you try to make the ϕ false, I can always make choices that force the end result to be true.
- If we were playing a game in which I was trying to make ϕ true while you were trying to make it false, and if I could always manage this, I'd be saying that I had a winning strategy.
- When we look at it this way, $\mathcal{PS}$-complete problems start to look like games – we take turns making moves, and we each try to reach our (mutually exclusive) winning conditions. So it'll not be too much of a surprise that a number of generalizations of two-player games are $\mathcal{PS}$-hard or $\mathcal{PS}$-complete.

This is why $\mathcal{PS}$ is important — $\mathcal{PS}$-completeness comes up when we're working in areas in which a program (like a robot) is competing against another agent. It also comes up when the robot is repeatedly taking measurements of its surroundings – our worst-case analysis treats the surroundings like an adversary, even if there is no actual adversary present.

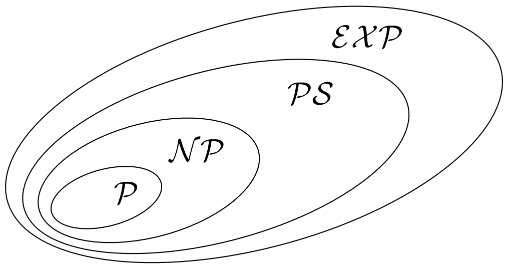
Space Complexity: What We've Covered:

Recall that $\mathcal{PS}$ is the class of languages that can be solved by some program that uses only a polynomial amount of memory. This class is interesting because it naturally expresses the types of problems that involve agents competing against each other – so many games are found in this class.

We showed that $\mathcal{NP} \subseteq \mathcal{PS}$, and that $\mathcal{PS} = \mathcal{NPS}$. So, all told, we know that

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS} = \mathcal{NPS} \subseteq \mathcal{EXP}.$$

We also believe that all of the classes are distinct:



That is, we believe (but have not proven) that more it is fundamentally easier to check solutions than to solve problems, and that bounding the time we have available is more restrictive than bounding the space we use.

We have actually proven one thing: We at least know that $\mathcal{P} \neq \mathcal{PS}$. See Theorem 9.10 in Sipser for details.

We finished by stating that the TQBF problem is $\mathcal{PS}$-complete, where

$$\text{TQBF} = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified boolean formula}\}.$$

Let's look a bit more at $\mathcal{PS}$-completeness. We've said that generalizations of many games are $\mathcal{PS}$-complete. These games include Hex, Reversi, and some versions of Go (when these games are played on arbitrarily large boards).

We should note that several one-player games such as Sokoban also have $\mathcal{PS}$-complete extensions.

To show these games are complete, we need to reduce to them from another $\mathcal{PS}$-complete problem. We'll give an example of one such reduction to a simple game: the generalized geography game.

Generalized Geography:

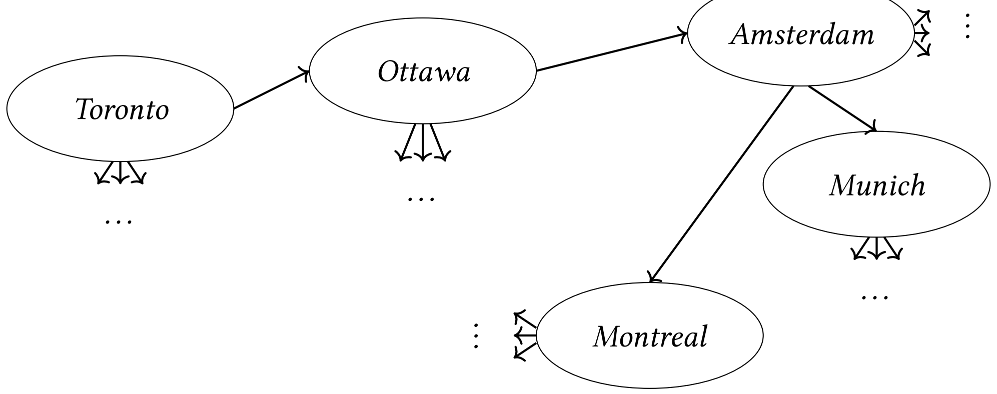
When playing *Geography*, players take turns naming cities. There are two rules:

1. No city can be named more than once.
2. The last letter of the current city named must be the first letter of the next.

E.g., *Toronto, Ottawa, Amsterdam, Munich, ...*

The first player who cannot continue the chain loses.

We can express this game by putting different city names on a graph:



That is, we draw an arrow between nodes u and v if the last letter of the city name in u is the first letter in v.

When using this graph to play geography, each player takes turns choosing the next node to visit: the next node must be reachable from the current one using a single edge, and no node can be visited twice.

Generalized Geography (GG) is the same, except we can play it on any directed graph G. In order to argue that this game is $\mathcal{PS}$-complete, we treat it as a language. So we formulate the game as:

$$\text{GG} = \left\{ \langle G, s \rangle \;\middle|\; \begin{array}{l} \text{Player A has a winning strategy for generalized} \\ \text{geography using } s \text{ as the starting vertex.} \end{array} \right\}$$

We argue that GG is $\mathcal{PS}$-complete – firstly, we know it is in PS since we can solve it using the following code:

$M =$ "On input $\langle G, s \rangle$:

1. If s has out-degree zero, reject, since player A loses.
2. Remove s and all edges containing s.
3. For each of the nodes v_1, v_2, …, v_k that s originally pointed to, recursively call M on $\langle G, v_i \rangle$.
4. If all recursive calls accept, player B has a winning strategy, so *reject.* Otherwise, player B doesn't have a winning strategy so player A must be able to force a win. So *accept.*

Question: How much space is required for this program?

There are at most ($n =$ the number of nodes in G) levels of recursion, and each step in the recursion just requires us to store a subgraph of G. So this is in $\mathcal{PS}$.

So to show that this is $\mathcal{PS}$-complete, we just need to show that TQBF $\leqslant_p$ GG.

Recall that this means that we want to be able to use GG to solve TQBF – if we have a quantified boolean formula ϕ we'd like to build, in polynomial time, a corresponding graph G with a starting node s. If we do it right, then determining if G and s are in GG would tell us whether ϕ is in TQBF.

Let's have a look at the ϕ from the previous class: $\phi = \forall x, y, \exists z, (x \vee z) \wedge (\overline{y} \vee z)$.

So how do we start?

We've already said that the quantifiers are a bit like player choices – one player chooses the variables under the $\forall$ quantifiers and tries to make the formula false, while another chooses the variables under the $\exists$ quantifiers and tries to make it true. We say that player A is trying to make the formula true.

So player B chooses x and y, and player A chooses z. A good place to start building our full graph would be to build a graph that makes the players go through a similar choice. One possible way to do this is to create nodes x, $\overline{x}$, and so on, in the graph, and connect them like so:

$$\begin{tikzcd}
 & & x \arrow[dr] & & y \arrow[dr] & & & z \arrow[dr] & \\
s \arrow[r] & v_1 \arrow[ur] \arrow[dr] & & v_2 \arrow[ur] \arrow[dr] & & v_3 \arrow[r] & v_4 \arrow[ur] \arrow[dr] & & v_5 \\
 & & \overline{x} \arrow[ur] & & \overline{y} \arrow[ur] & & & \overline{z} \arrow[ur] &
\end{tikzcd}$$

Why does this work?

- We want player A to choose z, and player B to choose x and y. Moreover, player B has to make the choices first, since that's the order in the quantifiers.
- If we look at the above graph, we see that player A starts on the s node, but has only one choice: the only place to go is v_1.
- Once we're at v_1, it's player B's turn. Once here, player B chooses either the x or the $\overline{x}$ node. This is like having player B choose to make x either true or false.
- Once player B makes a choice, it's player A's turn again – but there's only one place to go: v_2. At v_2, it's player B's turn again, and this time either y or $\overline{y}$ is set.
- After choosing y or $\overline{y}$, player A once again has no choice but to move to v_3. But once there, player 2 also has only one option: v_4.
- Once at v_4, player A gets to choose – and this time there's a choice between z or $\overline{z}$.
- Finally, player B is forced to move to v_5.

So this would be one way of building a graph that forces the players to choose their variable assignments. And the same process can be extended to longer chains of variables and quantifiers, too: we just repeat the diamond pattern above for every variable in the order it is seen in the formula. The diamond pattern forces a variable choice, and putting an extra step between diamonds changes the player making the choice.

So the players have made their choices. What about the rest of the formula?

We'll have the two player choose pieces of the formula while trying to get a true (or false) literal at the end. This is how the choices are made:

The formula we're using in our example is $(x \vee z) \wedge (\overline{y} \vee z)$. The players have made their variable choices, so player A wants to demonstrate that the variable choices make this true, and player B wants to show that it is false.

Since the formula is of the form $c_1 \wedge c_2$, the formula is true if both clauses are true, and false if at least one is false. One of the two players will choose a clause. Since player A would have to show all clauses are true and player B only needs to show that one is false, player B makes the decision. So player B selects one clause, say, $(\overline{y} \vee z)$.

No we have a smaller formula of the form $c_1 \vee c_2$. This formula is true if any of its clauses is true, and false if all of them are false. Again one of the two players will choose a clause. Since player B would have to show all clauses are false and player A only needs to show that one is true, player A makes the decision.

We can express this in the graph like so:

$$\begin{tikzcd}
 & & x \arrow[dr] & & y \arrow[dr] & & & z \arrow[dr] & & \\
s \arrow[r] & v_1 \arrow[ur] \arrow[dr] & & v_2 \arrow[ur] \arrow[dr] & & v_3 \arrow[r] & v_4 \arrow[ur] \arrow[dr] & & v_5 \arrow[r] & v_6 \arrow[ddll] \arrow[ddddll] \\
 & & \overline{x} \arrow[ur] & & \overline{y} \arrow[ur] & & & \overline{z} \arrow[ur] & & \\
 & & & & & z & & c_2 = (\overline{y} \lor z) \arrow[ll] \arrow[dll] & & \\
 & & & & & \overline{y} & & & & \\
 & & & & & z & & c_1 = (x \lor z) \arrow[ll] \arrow[dll] & & \\
 & & & & & x & & & &
\end{tikzcd}$$

So we've added nodes to the graph for every clause and subclause in the formula. If the clauses are joined by *and*s, player B chooses which clause to go to. If they are joined by *or*s, player A chooses. If there is a *not*, the role of the players changes – player A wants to make the clause false. So we put another step in the graph so make the players switch turns.

Finally, we want to finish the game. We should do this in a way such that if player A wins if the final literal is true, and player B wins otherwise (aside from the role switching through *not*s).

We do this by connecting the endpoints of the graph back to the nodes in the variable choices like so:

$$\begin{tikzcd}
 & & x \arrow[dr] & & y \arrow[dr] & & & z \arrow[dr] & & \\
s \arrow[r] & v_1 \arrow[ur] \arrow[dr] & & v_2 \arrow[ur] \arrow[dr] & & v_3 \arrow[r] & v_4 \arrow[ur] \arrow[dr] & & v_5 \arrow[r] & v_6 \arrow[ddll] \arrow[ddddll] \\
 & & \overline{x} \arrow[ur] & & \overline{y} \arrow[ur] & & & \overline{z} \arrow[ur] & & \\
 & & & & & z \arrow[uuurr] & & c_2 = (\overline{y} \lor z) \arrow[ll] \arrow[dll] & & \\
 & & & & & \overline{y} \arrow[uul, bend left] & & & & \\
 & & & & & z \arrow[uuuuurr] & & c_1 = (x \lor z) \arrow[ll] \arrow[dll] & & \\
 & & & & & x \arrow[uuuuuulll, bend left] & & & &
\end{tikzcd}$$

For this equation, we see that player A chose the literal, so player B has to make the next step. We've just chosen the next arrow to make it possible for there to be one more step iff that literal was false. The process would be similar if player A had the choice at this point.

And that's the idea behind the reduction – we can see that the same idea will work for any formula ϕ, and that the construction is polytime in ϕ. So TQBF $\leqslant_p$ GG, and so GG is $\mathcal{PS}$-complete.

Logarithmic Space:

The other space complexity class we'll talk about is the class of logarithmic space problems – that is, the class of problems that take an amount of space that's logarithmic in the size of the input.

Since the input already takes $\mathcal{O}(n)$ space, though, we can't include it in the memory being counted. So we consider only TMs that have a read-only input tape, a logarithmic sized work tape, and a write-only output tape. (The inputs and outputs have to restricted, otherwise we could cheat and get a linear amount of memory).

Definition: $\mathcal{L} = \{L|L \text{ is decided by a TM that operates in } \mathcal{O}(\log n) \text{ space}\}$.

Definition: $\mathcal{NL} = \{L|L \text{ is decided by an NTM that operates in } \mathcal{O}(\log n) \text{ space}\}$.

Important: this says $\mathcal{O}(\log n)$, not $\mathcal{O}(\log^k n)$.

Question: *Why do we care about this class?*

The idea is that we might be running a machine M that is itself small (limited memory), but which has access to a very large database D. In order to access D, M must be able to say where on D it wants to look. So if D has 2^n entries, M must be able to store $\log(2^n) = n$ numbers.

This type of application comes up when, say, M is your phone or laptop and D is the Internet.

It's because we're looking at this sort of application that we also have the restriction that we work in a space that's $\mathcal{O} \log n$ and not, say, $\log^2(n)$. (Also because $\log(n^k) = k \log n$).

This is a very restricted class – there's really not much we know how to do when we can't even copy the input to the working tape. But we can do some things: we can add and multiply numbers, and we can decide simple languages like $\{0^n 1^n | n \in \mathbb{N}\}$.

How to decide $\{0^n 1^n | n \in \mathbb{N}\}$ in logspace: we can't write to the input tape, and we can't copy the input. What we can do is count the zeros and ones, since the space needed to remember a number n is logarithmic in n. So our algorithm would look like:

On input x:

```
1. 1. Set a counter $c$ to 0.
2. While the tape head reads a 0:
3.     Increment $c$ by 1 and move the head right.
4. While the tape head reads a 1:
5.     Decrement $c$ by 1 and move the head right.
6. If $c$ is set to 0 as the end of the input is reached, *accept.* If it reaches 0 before that time,
   or if it is non-zero at the end of the input, *reject.*
```

An important consequence of the fact that $\mathcal{L}$ uses $\mathcal{O}\log n$ and not $\mathcal{O}\log^2(n)$ space is that Savitch's theorem doesn't give us a way to emulate non-determinism and still stay in the class. So the best we can currently say is that $\mathcal{L} \subseteq \mathcal{NL}$. We believe that the two classes are not equal.

So what we know is that

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PS} \subseteq \mathcal{EXP}.$$

We believe all of the inequalities are strict, but aside from the facts that $\mathcal{L} \neq \mathcal{PS}$ and $\mathcal{P} \neq \mathcal{EXP}$, we don't yet have any proofs. We haven't even proven that 3SAT $\notin \mathcal{NL}$!

For reference, if you want to better understand the class $\mathcal{NL}$, we're going to need to do reductions again. But this time we're looking at a class that's contained in $\mathcal{P}$, so even polytime reductions would be too powerful. So we do logspace reductions:

Definition: We say that $A \leqslant_L B$ is there is a function f computable in logspace such that $x \in A$ iff $f(x) \in B$.

If we're working in logspace we won't, in general, even have the space to write $f(x)$. But the point of reductions is to argue that B can be used to solve A. So when testing whether if $f(x)$ is a member of B in order to tell us whether x is in A, we've got to test whether $f(x) \in B$. When we do so, we have to call the function f repeatedly.

Computations in $\mathcal{L}$ and $\mathcal{NL}$ often involve repeating the same calculation over and over in order to save space.

Definition: A language L is $\mathcal{NL}$-complete if it is in $\mathcal{NL}$, and if for every language A in $\mathcal{NL}$, $A \leqslant_L L$.

Our go-to $\mathcal{NL}$-complete language is PATH, where

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}.$$

We've seen this language before. This is like asking if you can get from one page in Wikipedia to another by following links.

We won't prove this fact, but you can see Sipser 8.25 for details.

Finally, we can show that PATH $\in$co-$\mathcal{NL}$! Again, we won't prove this fact here.
So $\mathcal{NL}$ =co-$\mathcal{NL}$.

That concludes our introduction to logspace (and to space complexity).

Over the next two weeks, we'll shift the type of problems we look at a little bit. Up to now we've mostly looked at yes-no questions (decision problems), but next week we'll look at the types of functions that TMs can efficiently compute.

The Take-Away from this Lesson:

- We described the Cook-Levin theorem and showed that 3SAT is $\mathcal{NP}$-complete.
- We introduced space complexity and the classes $\mathcal{PS}$ and $\mathcal{NPS}$.
- We talked about Savtitch's theorem and the fact that $\mathcal{PS} = \mathcal{NPS}$.
- We discussed reductions in $\mathcal{PS}$ and argued that there is a $\mathcal{PS}$-complete language, TQBF.
- We introduced the Generalized Geography game, and showed that it is $\mathcal{PS}$-complete.
- We briefly talked about the class of languages that can be decided using a logarithmic amount of space.

Glossary:

$\mathcal{L}$, Logspace reductions, $\mathcal{NL}$, $\mathcal{NPS}$, NSPACE($f(n)$), $\mathcal{PS}$, Savitch's Theorem, SPACE($f(n)$)

***Languages*:**

GG, TQBF