CSCC63 – Week 10

This week: We'll finish talking about space complexity, then we'll talk about some of the functions that TMs can compute.

Space Complexity, Continued

Last week we described space complexity and introduced the class $\mathcal{PS}$, as well as the idea of $\mathcal{PS}$-completeness.

Recall that we know a $\mathcal{PS}$-complete language: the True Quantified Boolean Formula language (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 (we won't go over this in too much detail: we've seen the the two big parts of it already).

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 of that idea.*

We can treat the languages in $\mathcal{PS}$ as games, where each quantifier reflects a choice made by one of the players (in contrast, we can treat the languages in $\mathcal{NP}$ as puzzles, where the certificate is the puzzle solution). So many space-limited games turn out to be $\mathcal{PS}$-complete. These games include Hex, Reversi, and some versions of Go[1] (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.

[1] If you're interested, there is a note in the Other folder of the course website on the computationl complexity of Go. This includes a reduction from a version of the Generalized Geography game that we look at today.

$\mathcal{PS}$ and Games: 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 $\mathcal{PS}$-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.

Generalized Geography:

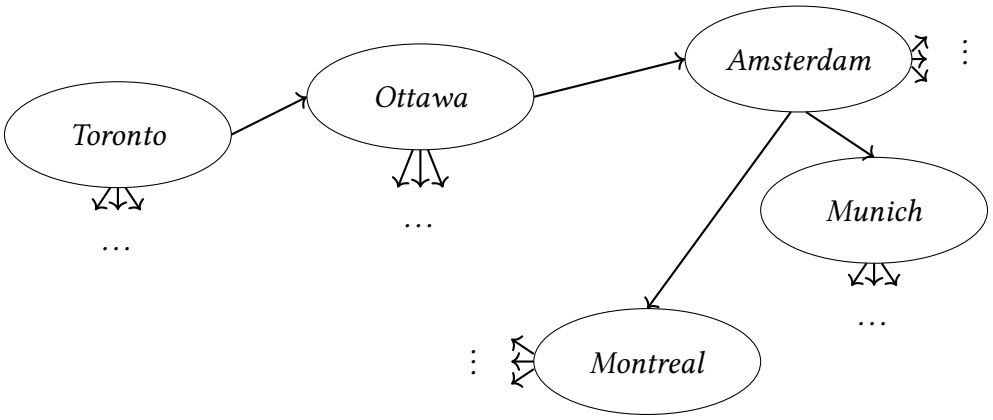
When playing *Geography*, players take turns naming cities. There are two rules:

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 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 $\mathcal{PS}$ 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.

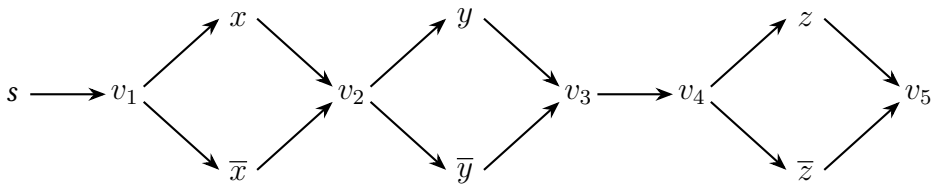
Recall that this means that we want to be able to use GG to solve TQBF – if we have a quantified boolean formula ϕ we'd like to build, in polynomial time, a corresponding graph G with a starting node s. If we do it right, then determining if G and s are in GG would tell us whether ϕ is in TQBF.

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:



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?

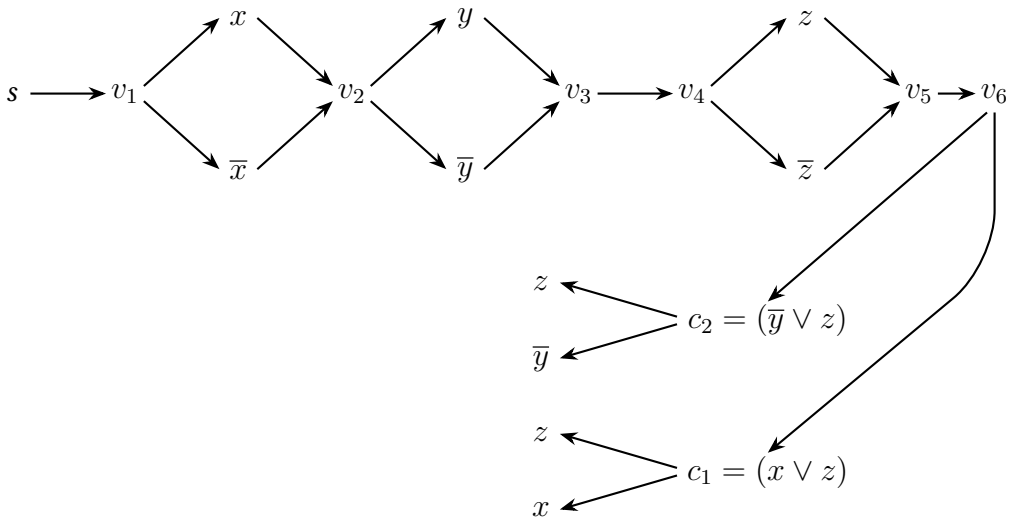
We'll have the two player choose pieces of the formula while trying to get a true (or false) literal at the end. This is how the choices are made:

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.

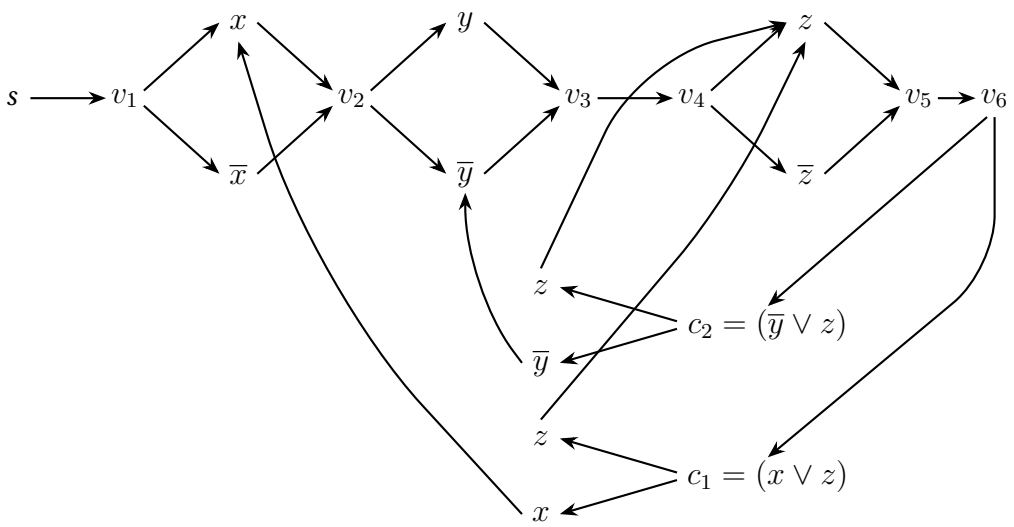
We can express this in the graph like so:



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:



For this equation, we see that player A chose the literal, so player B has to make the nest step. We've just chosen the next arrow to make it possible for there to be one more step iff that literal was false.

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 $\mathcal{O}(\log^k n)$.

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 $\{0^n 1^n | n \in \mathbb{N}\}$.

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 $\log(n^k) = k\log n$).

How to decide $\{0^n1^n | n \in \mathbb{N}\}$ 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 $\text{3SAT} \notin \mathcal{NL}$!

If we were to try to better understand the class $\mathcal{NL}$, we'd 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 $\mathcal{NL}$, $A \leq_L L$.

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

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

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

The language PATH is clearly in $\mathcal{NL}$: you can just start at s and non-deterministically follow edges until you reach t. You can halt after checking n vertices, where n is the number of vertices in G.

The proof that $\forall A \in \mathcal{NL}, A \leqslant_L \text{PATH}$ is more complicated. Like Savitch's theorem, we won't cover this in detail, and we won't be expecting you to remember this proof, but it is good to know. You can see Sipser 8.25 for details.

The basic idea is that if $A \in \mathcal{NL}$, it is decided by a logspace NTM. We can treat configurations of this NTM as the nodes in a graph. If we do so, the question of whether an accepting configuration can be reached becomes a graph reachability problem, which is really what PATH decides.

Finally, we can show that PATH $\in$ co-$\mathcal{NL}$! Again, we'll only gloss over the proof, but the details can be found in Sipser 8.27.

What this means os that $\mathcal{NL} =$ co-$\mathcal{NL}$.

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

Now, let's look at something different: up until now we've only relly looked at decision problems – quetions whose answers are always yes or no. But we may want to work with functions whose outputs are numbers, graphs, or other types. So let's look at the functions that TMs can efficiently compute.

Computable Functions

As we've said, sometimes we want a number or a string or some other more complex data type as a return value. How can we talk about these computations?

> E.g., instead of asking "Is $x^2 = y$?" we could ask "What is x^2?".
>
> Instead of asking "Does the graph G have a path from s to t?" we could ask
>
> - "What is a path from s to t?", or
> - "What is the longest/shortest path from s to t?", or even
> - "How many s to t paths are there?".
>
> Each of these questions ends up being very different.

As a recap, the return value of a TM is whatever is on its tape *(or on its output tape, if it has a designated output tape)* when it halts. Practically we just indicate this by adding a *return* line to our pseudocode, however.

There's one type of non-decision computation we've been using throughout the course: all of our reductions are non-decision computations, if you think about it …

Now, the first thing to remember is that the problems in this domain are, by definition, not *yes/no* problems. The decidable and recognizable language classes, $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{PS}$ — any language class we care about — don't contain these problems. It's not an issue of difficulty, though, it's more that it's an apples to oranges comparison.

But there are functional analogues to these language classes (e.g., the analogue to the class of decidable languages would be the class of computable functions). We'll look at those classes here.

With that in mind, the first class of functions that we'll look at is the class of functions that can be computed in polynomial time. This is the functional analogue to $\mathcal{P}$, and we refer to it as $\mathcal{FP}$.

The class $\mathcal{FP}$ contains all of our polytime reductions, as well as questions like

- "Given an x, what is x^2?",
- "Given a graph G and an s and t, is there a path in G from s to t?", or
- "Given a graph G and an s and t, is what is the shortest path in G from s to t?".

But if we have a $\mathcal{P}$ analogue, could we find an $\mathcal{NP}$ analogue?

Actually, yes — consider the follwing problem:

FIND-HAM-PATH:

> **Input:** A graph G with specified vertices s and t.
>
> **Output:** Find a Hamiltonian path in G from s to t. Return *null* if no such path exists.

A language L is in $\mathcal{NP}$ if we can describe it in terms of its certificates — if there's some polytime verifier V such that

$$L = \{x \mid \exists c, V \text{ accepts } \langle x, c\rangle\}.$$

a very natural $\mathcal{NP}$-analogue question would be: "Given an x, find one of its certificates c (or *null* if no certificate exists)".

We refer to the types of problems that can be expressed this way as $\mathcal{FNP}$. This class is a close analogue to $\mathcal{NP}$: in particular:

> $\mathcal{FP} \subseteq \mathcal{FNP}$, and we strongly suspect that these classes are different.
>
> *In fact,* $\mathcal{P} = \mathcal{NP}$ *iff* $\mathcal{FP} = \mathcal{FNP}$.

There is a very interesting fact about $\mathcal{FNP}$ to remember:

Suppose we take some language L that is in $\mathcal{NP}$. This L will have an associated $\mathcal{FNP}$ problem f_L. If L is, in fact, $\mathcal{NP}$-complete, it will always be the case that we can use a solver for L to solve f_L, as well. We say that L is *self-reducible*.

Let's see an example. Suppose we had an oracle **HP** that could solve the decision problem HAM-PATH for us: given $\langle G, s, t\rangle$, **HP** would return *true* if G has a Hamiltonian path from s to t, and *false* otherwise.

We can use this program to solve FIND-HAM-PATH as follows:

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

1. Query **HP**($\langle G, s, t\rangle$). If it returns false, return *null*.
2. Let $G' = G$.
3. For every edge e in G':
4. Let $G'' = G'$ with e removed.
5. Query **HP**($\langle G'', s, t\rangle$).
6. If it returns true, set $G' = G''$.
7. Set $v_1 = s$, and use the remaining edges in G' to give an ordering to the vertices

A Sample Justification:

If there is no Hamiltonian path in G, we return *null* in line 1, as required.

Otherwise, we know there is some Hamiltonian path in G.

Let $P[i]$ the predicate that is true iff, after i iterations of the loop from steps 3 to 6:

i) G' is a subgraph of G, and

ii) G' has a Hamiltonian path from s to t.

Base Case: Let $i = 0$.

Then after i loop iterations, $G' = G$, and so property i) holds.

Furthermore, since we are considering the case where we haven't returned *null* in line 1, we know that $G = G'$ has a Hamiltonian path from s to t. So property ii) holds.

Induction Step: Suppose $P[i]$ holds for some i [IH].

Then, after i loop iterations,

i) G' is a subugraph of G, and

ii) G' has a Hamiltonian path from s to t.

Now, consider the program state after $i + 1$ loop iterations. Let $\hat{G'}$ be the value of G' after the $(i + 1)^{st}$ loop iteration:

Case 1: If **HP** returns false in line 5 of the $(i + 1)^{st}$ loop iteration, then G' remains unchanged ($\hat{G'} = G'$). So properties i) and ii) hold after $i + 1$ loop iterations.

Case 2: If **HP** returns true in line 5 of the $(i + 1)^{st}$ loop iteration, then $\hat{G'}$ is set to G'', where G'' is G' with one edge removed.

i) Since G'' is G' with one edge removed, $\hat{G'}$ is a subgraph of G', which is itself a subgraph of G. So property i) holds.

ii) Since **HP** has returned true when queries on $\langle G'', s, t\rangle$, we know that G'' has a Hamiltonian path from s to t. So property ii) holds.

So we can see that properties i) and ii) hold after every loop iteration, and so for all i, $P[i]$ holds after i loop iterations.

Now, the loop must terminate, since there is only one iteration per edge in G, and since the number of edges in G is finite.

So the graph G' after the loop has finished is a subgraph of G that contains a Hamiltonian path. We'll call this path H.

Since M never adds edges to G', this H must have been present in G' after every loop iteration.

Now, consider any edge e in G that is not in H. There must be some loop iteration in which M tried to remove e from G'. By our previous argument, H was a subgraph of G', and so **HP** must have returned true.

So the edge e must have been removed from G' in line 6.

Therefore, the G' at the end of the loop contains only the edges in H.

By using these edges in line 7 to determine the order of the vertices in H, we return a Hamiltonian path, as required.

As another example, we can use an oracle **3S** for 3SAT to find a satisfying assignment, if one exists, as well. Consider the following:
FIND-SATISFYING-ASSIGNMENT:

> **Input:** An input 3CNF ϕ.
>
> **Output:** A satisfying truth assignment τ for ϕ. Return *null* if no such truth assignment exists.

We can solve this problem with the following program:

Let $M =$ "On input $\langle \phi \rangle$:

1. Run **3S**($\langle \phi \rangle$). If it returns false, return null.
2. Let $\phi' = \phi$ and $\tau = \{\}$.
3. For every variable x_i in ϕ':
4. Let $\phi'' = \phi' \wedge (x_i \vee x_i \vee x_i)$.
5. Run **3S**($\langle \phi'' \rangle$).
6. If it returns true, set $\phi' = G''$ and $\tau = \tau \cup \{x_i : \text{TRUE}\}$.
7. Otherwise, set $\phi' = G' \vee (\overline{x_i} \vee \overline{x_i} \vee \overline{x_i})$ and $\tau = \tau \cup \{x_i : \text{FALSE}\}$.
8. Return τ."

In Fact...

We know that we can do this for *any* $\mathcal{NP}$-complete language!

> *Why?*

Suppose that L is some $\mathcal{NP}$-complete language. We know from last week (specifically, from the Cook-Levin theorem) that there is a polytime reduction $L \leqslant_p$ 3SAT. Let's look at the reduction we've seen in class.

Since L is $\mathcal{NP}$-complete it has a verifier V. We'll use this V to write the following NTM:

> Let $N =$ "On input $\langle x \rangle$:
>
> 1. Nondeterministically choose certificate c for x.
> 2. Deterministically run V on $\langle x, c \rangle$.
> 3. If it accepts, *accept*. Otherwise, *reject*.

If we use this N as our NTM, look at what happens if we go through the Cook-Levin reduction. As you my recall, if we're given an instance $\langle x \rangle$ of L, we try to build a table contining the configurations that N takes on an accepting run on x:

#	q_0	x_0	x_1	...	x_{n-1}	␣	␣	...	␣	#
#	$t_{1,0}$	q_1	x_1	...	x_{n-1}	␣	␣	...	␣	#
#										#
#										#
⋮	⋮	⋮			⋮					⋮
#	$t_{n^k-1,0}$	$t_{n^k-1,1}$			...				t_{n^k-1,n^k-1}	#

The 3CNF ϕ that we build encodes this table and enforces the constraints that ensure that the table contents are really the TM configurations.

> *And almost all of our variables directly state what's in the table cells... Our Boolean variables tell us, e.g.,* yes/no*, does the cell at position* $(100, 51)$ *contain a* 0*?*
>
> *There are a few extra variables we add right at the end to get the 3CNF form we want, but they're extra – we can almost think of them as something like a parity check.*

This mens that if I have an accepting sequence of configurations I can find a satisfying truth assignment, but it also means I can go the other way: If I have a satisfying truth assignment τ for the resulting 3CNF ϕ, I can use it to refill the table.

> *And we've chosen to use an NTM* N *that explicitly writes the certificate* c *that it uses. So if we can find a* τ*, we can refill the table, and we can find certificate* c *for* x*.*

So finding a certificate for x boils down to finding a certificate for ϕ. Can we do this?

> *Sort of: this is just the* FIND-SATISFYING-ASSIGNMENT *problem we've already solved. So if we had an orcle for* 3SAT *we could find a certificate for* x *in polytime.*

But what if we don't have an oracle for 3SAT? Wht if we only have an oracle for L?

If L is $\mathcal{NP}$-complete, though, we have a polytime reduction $f : \text{3SAT} \leqslant_p L$. We can use that to simulate a 3SAT oracle, since every time we want to test whether a $\phi' \in$ 3SAT we can just find $f(\langle \phi' \rangle)$ and feed that into the L oracle.

So to find certificate for x, we just run it through the Cook-Levin reduction to get a ϕ, and then we run our FIND-SATISFYING-ASSIGNMENT solver on ϕ, using our 3SAT to L reduction and our L oracle in plce of the 3SAT oracle. If you walk through the steps, you'll see the whole proces is polytime.

Any calcultion of this form – where we use an oracle for L to find a certificate – is referred to as a **(downward) self-reduction**. You can expect to see a few of these in your assignments and tests.

A Restriction for Class

In your tests and assignments, when you're asked to find a self-reduction, you may not directly use the Cook-Levin reduction unless otherwise stated.

Note that the class $\mathcal{FNP}$ is a bit stronger than $\mathcal{NP}$: if we run a Hamiltonian path finder on $\langle G, s, t\rangle$ and find that there is no Hamiltonian path, then we would have identified a *no*-instnace of HAM-PATH. So we could solve the co-$\mathcal{NP}$-complete problem $\overline{\text{HAM-PATH}}$ as well as the $\mathcal{NP}$-complete problem HAM-PATH.

The Take-Away from this Lesson:

- 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.
- We've briefly described the major function classes $\mathcal{FP}$ and $\mathcal{FNP}$.
- We've described how to use an oracle for the decision versions of HAM-PATH and 3SAT to quickly find one of their certificates.
- We've sketched a proof as to why this can be done for any $\mathcal{NP}$-complete language.

Glossary:

$\mathcal{FP}$, $\mathcal{FNP}$ $\mathcal{L}$, Logspace reductions, $\mathcal{NL}$

***Problems*:**

FIND-HAM-PATH, FIND-SATISFYING-ASSIGNMENT, GG, TQBF