Reminder

- In class we introduced space-complexity, defined as the maximum number of tape cells used by a Turing machine (on any computational branch for non-deterministic machines).
- PSPACE = $\bigcup_k \text{SPACE}(n^k)$.
- NPSPACE = $\bigcup_k \text{NSPACE}(n^k)$.
- Savitch’s Theorem:
  - Statement: For any function $f: \mathbb{N} \to \mathbb{R}^+$ where $f(n) \geq n$, we have that $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.
  - Corollary: PSPACE = NPSPACE.
- PSPACE-completeness: A language $B$ is PSPACE-complete if:
  1. $B \in \text{PSPACE}$ (PSPACE membership)
  2. $A \leq_P B \; \forall A \in \text{PSPACE}$ (PSPACE-hardness)

Why do we do poly-time (P) reductions?

When showing a language is NP-complete or PSPACE-complete we’ve used polynomial time reductions. Why don’t we use NP reductions or PSPACE reductions instead?

Let’s go over the basics of reductions again. In general, a reduction from some language $A$ to some language $B$ is a function $f: \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*$, $w \in A \iff f(w) \in B$. If we have $A \leq B$, this tells roughly that solving $B$ is at least as hard as solving $A$.

For mapping-reducibility we added the condition that $f$ be Turing-computable. Notice that for mapping-reducibility, all (except for $\Sigma^*$ and $\emptyset$) decidable problems are reducible to each other. Essentially, given that $B \neq \Sigma^*, \emptyset$ we know that there exists $x, y \in \Sigma^*$ such that $x \in B$ and $y \notin B$. Then our reduction is simply

$$f(w) = \begin{cases} \ x & w \in A \\ \ y & w \notin A \end{cases}$$

Computing $f(w)$ simply involves using the decider for $A$, so the reduction is Turing-computable.

For polynomial-reducibility we added the condition that $f$ be computable in polynomial time. Similarly for polynomial-reducibility, all (except for $\Sigma^*$ and $\emptyset$) problems in P are reducible to each other.
In fact, given some class of languages C, all non-trivial languages in C are C-reducible to each other; the reduction simply solves the problem in question and produces an instance with the same value. This means that “completeness” is only meaningful if you use a class of reductions from a potentially smaller (or weaker) class. Take PSPACE for example; it wouldn’t make any sense to do PSPACE reductions. You could potentially use other reductions, however in practice our reductions are simple transformations that are easily doable in polynomial-time.

By convention, we use polynomial-time reductions for classes within the polynomial hierarchy. However, when studying classes contained within P we will use log-space reductions, as we will see later on in the course.

Example 1 — TBQF is PSPACE-complete

Recall that TQBF = \{⟨φ⟩ | φ is a true fully-quantified boolean formula\}. Show that TQBF is PSPACE-complete.

Solution

First, we need to show that TQBF ∈ PSPACE.

Given input ⟨φ⟩ ...

- If φ contains no quantifiers, then it is an expression with only constants, so evaluate φ and accept if it is true; otherwise, reject.
- If φ = ∃x ψ for some formula ψ, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then accept; otherwise, reject.
- If φ = ∀x ψ for some formula ψ, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then accept; otherwise, reject.

We need to show that our algorithm runs in polynomial space. Note that the depth of the recursion is at most the number of variables. At each level we need only store the value of one variable, so the total space used is O(n), where n is the number of variables that appear in φ.

Now let A be a language decidable by some Turing machine M in space n^k for some constant k. To show that TQBF ∈ PSPACE, we will show that A ≤_p TQBF. Given some w, we want to construct φ such that w ∈ A ⇐⇒ φ ∈ TQBF.

Let each configuration of M have a variable for each tape symbol and state, corresponding to their possible settings. Then each configuration has n^k cells and so is encoded by O(n^k) variables. Given two configurations c_a, c_b representing two collections of variables and some integer t > 0, we want to create a formula φ_{a,b,t} such that φ_{a,b,t} is true iff M can go from c_a to c_b in at most t steps. The idea is to then have φ = φ_{start,accept,h}, where h = 2^{(dn^k)} with d chosen such that M has no more
than $2^{d n^k}$ possible configurations on an input of length $n$.

For $t = 1$, we design $\phi_{a,b,1}$ to evaluate to true only if $c_a = c_b$ or $c_b$ follows from $c_a$ in a single step of $M$. For the first possibility, we express the equality by writing a boolean expression saying that each of the variables representing $c_a$ contains the same Boolean value as the corresponding variable representing $c_b$. For the second possibility, we can express that $c_a$ yields $c_b$ in a single step of $M$ by writing boolean expressions stating that the contents of each triple of $c_a$’s cells correctly yields the contents of the corresponding triple of $c_b$’s cells.

For $t > 1$, we recursively construct $\phi_{a,b,t}$. Notice that if we can go from $c_a$ to $c_b$ in at most $t$ steps, then there exists some configuration $c_d$ such that we can go from $c_a$ to $c_d$ and $c_d$ to $c_b$ in at most $t/2$ steps each. One possible way to write this formula is $\phi_{a,b,t} = \exists d [\phi_{a,d,t/2} \land \phi_{d,b,t/2}]$. However, the size of the formula doubles at each stage of the recursion, which for $t = h$ gives us a formula of size $2^{2(n^k)}$, which is exponential with respect to $n$. Alternatively, we can write the equivalent formula $\phi_{a,b,t} = \exists d \forall u \forall v ((u = a \land v = d) \lor (u = d \land v = b)) [\phi_{a,v,t/2}]$.

Finally, we need to show our reduction is in polynomial time. Each stage of the recursion runs in $O(n^k)$, and the maximum recursion depth is $\log_2 h = O(n^k)$. Thus our reduction runs in $O(n^2 k)$ time, and therefore we have $A \leq_P \text{TQBF}$. We conclude that $\text{TQBF}$ is PSPACE-complete.

Example 2 — $\neq \text{SAT}$ is NP-complete (Problem 7.26)

Let $\phi$ be a 3cn-formula. An $\neq$-assignment to the variables of $\phi$ is one where each clause contains two literals with unequal truth values. In other words, an $\neq$-assignment satisfies $\phi$ without assigning three true literals in any clause.

a) Show that the negation of any $\neq$-assignment to $\phi$ is also a $\neq$-assignment.

Notice that for a $\neq$-assignment to be valid, it means that for each clause in the formula there is at least one literal being true and one literal being false. Negating the whole $\neq$-assignment assignment ensures that each true literal becomes false and each false literal becomes true. This clearly preserves the invariant that each clause has at least one false and at least one true literal.

b) Let $\neq \text{SAT}$ be the collection of 3cnf-formulas that have an $\neq$-assignment. Show that we obtain a polynomial time reduction from $3\text{SAT}$ to $\neq \text{SAT}$ by replacing each clause $c_i$

$$(y_1 \lor y_2 \lor y_3)$$

with the two clauses

$$(y_1 \lor y_2 \lor z_i) \land (\overline{z_i} \lor y_3 \lor b)$$
where \( z_i \) is a new variable for each clause \( c_i \), and \( b \) is a single additional new variable.

\[ \Rightarrow \]

We first show that if the original formula \( \phi \) from 3SAT has a satisfying assignment, then the resulting formula \( \phi' \) (given by the reduction) in \( \neq SAT \) is also satisfiable. That is, given a satisfying assignment for \( \phi \), we build a satisfying \( \neq \)-assignment for \( \phi' \).

For each variable in the satisfying assignment of \( \phi \) we also set the same value for that variable in the assignment for \( \phi' \) (e.g. if the satisfying assignment for \( \phi \) set \( x \to T, y \to F \) then we also set \( x \to T, y \to F \) in the satisfying assignment for \( \phi' \)).

Now all that is left to do is assign values to \( z_i \) for each \( i \) and also choose a value for \( b \). For each clause \( c_i \ (y_1 \lor y_2 \lor y_3) \) in \( \phi \), set \( z_i \) to false if any of \( y_1 \) and \( y_2 \) (or both) are true. If both \( y_1 \) and \( y_2 \) are false (meaning \( y_3 \) must be true), then set \( z_i \) to true.

Finally, set \( b \) to false. Let’s now have a look at why this works. Remember, we are replacing each clause \( c_i \)

\[
(y_1 \lor y_2 \lor y_3)
\]

with the two clauses

\[
(y_1 \lor y_2 \lor z_i) \land (\overline{z_i} \lor y_3 \lor b)
\]

1. If any (or both) of \( y_1 \) and \( y_2 \) are true, then the two new clauses in \( \phi' \) will be satisfied because we are setting \( z_i \) to false. This satisfies the first of the two new clauses, and the second one as well (\( \overline{z_i} \) will be true and \( b \) is set to false).

2. If both \( y_1 \) and \( y_2 \) are false (meaning \( y_3 \) is true), we are setting \( z_i \) to true to satisfy the first of the two new clauses. The second new clause will be satisfied as well (\( \overline{z_i} \) will be false and \( y_3 \) is true).

Remember that for each satisfying \( \neq \)-assignment there is another satisfying assignment obtained through the negation of all the variables! This means there is (at least) another satisfying assignment which always binds \( b \) to true. The variable \( b \) here is just a helper variable and given the nature of \( \neq \)-assignments it doesn’t really matter what value it takes.

\[ \Leftarrow \]

We now show that given a satisfying \( \neq \)-assignment for \( \phi' \), we can obtain a satisfying assignment for \( \phi \).

First, if \( b \) is set to true, then we negate all the variables in the \( \neq \)-assignment. This ensures \( b \) is now set to false. Notice that when we set \( b \) to false, we ensure that at least one of \( y_1 \), \( y_2 \) and \( y_3 \) is true for each pair of clauses.

Then, we just ignore every \( z_i \) and \( b \), and build the assignment for \( \phi \) by just looking at the variables in the \( \neq \)-assignment for \( \phi' \). Again, we have ensured that at least one of \( y_1 \), \( y_2 \) and \( y_3 \) is true, meaning that the each clause in \( \phi \) will be satisfied.
c) Conclude that \( \neq SAT \) is NP-complete.

We start by observing that \( \neq SAT \) is in NP. Similarly to any other SAT problem, a NTM can guess the correct assignment in polynomial time. Furthermore, in part b) we have showed that \( \neq SAT \) is NP-hard by providing a reduction from 3SAT to \( \neq SAT \). The reduction (building clauses) is trivially polynomial. Since \( \neq SAT \) is both in NP and NP-hard, then \( \neq SAT \) is NP-complete.

**Example 3 — MAX-CUT is NP-complete (Problem 7.27)**

A cut in an undirected graph is a separation of the vertices \( V \) into two disjoint subsets \( S \) and \( T \). The size of a cut is the number of edges that have one endpoint in \( S \) and the other in \( T \). Let

\[
MAX-CUT = \{ (G, k) \mid G \text{ has a cut of size } k \text{ or more} \}
\]

Show that \( MAX-CUT \) is NP-complete. (Hint: Show that \( \neq SAT \) polynomially reduces to \( MAX-CUT \). The variable gadget for variable \( x \) is a collection of \( 3c \) nodes labeled with \( x \) and another \( 3c \) nodes labeled with \( \overline{x} \), where \( c \) is the number of clauses. All nodes labeled \( x \) are connected with all nodes labeled \( \overline{x} \). The clause gadget is a triangle of three edges connecting three nodes labeled with the literals appearing in the clause. Do not use the same node in more than one clause gadget.)

**Solution**

\( MAX-CUT \) is in NP because a deterministic Turing machine can check a solution in polynomial time, where the certificate in this case is the subset of nodes that form the cut.

We now reduce \( \neq SAT \) to \( MAX-CUT \). We are going from a formula \( \phi \) (with \( n \) variables and \( c \) clauses) to an undirected graph \( G \) and a number \( k \). As the hint suggests, for each variable \( x \) in \( \phi \), we add \( 3c \) nodes labeled with \( x \) and \( 3c \) nodes labeled with \( \overline{x} \) to \( G \). We connect all "positive" nodes for a certain variable to all the "negative" nodes for that same variable (e.g. all \( x \) nodes are connected to all \( \overline{x} \) nodes). For each variable there are therefore \( (3c)^2 \) edges. Then, for each clause in \( \phi \), we pick three nodes from the respective literals (e.g. if the clause is \( a \lor b \lor \overline{c} \), we pick a node labeled with \( a \), one labeled with \( b \) and one labeled with \( \overline{c} \)) and we add edges between them, forming a triangle. We make sure to never reuse a node for this purpose. Finally, we set \( k = n(3c)^2 + 2c \).

We now show that this reduction works.

\[ \Rightarrow \]

We start from a satisfying \( \neq \)-assignment of \( \phi \) and we build a cut for \( G \) of size \( k \) or more. We simulate setting variables to true by putting all the respective literal nodes in the cut. For example, if variable \( x \) is set to true in the \( \neq \)-assignment of \( \phi \), we put all nodes labeled \( x \) in the cut subset. Similarly, if \( x \) is set to false, we put all nodes labeled \( \overline{x} \) in the cut subset instead.

Notice that putting all nodes of a certain literal in the subset makes it so that exactly \( (3c)^2 \) edges are now in the cut (all the edges going from all the nodes \( x \) to all the nodes \( \overline{x} \) where \( x \) is the literal). If we do this for each variable and there are \( n \) variables, we end up with \( n(3c)^2 \) edges in the cut.
Now, we observe that for each clause, since it is satisfied, at least one literal node will be out of the cut subset (because it needs to be false). This ensures that for each clause, there are two additional edges that will be part of the cut. If there are \( c \) clauses, we end up with \( 2c \) edges in the cut.

This means the total number of edges in the cut (the size of the cut) will be \( n(3c)^2 + 2c \), which is equal to \( k \). Therefore, our constructed \( \langle G, k \rangle \) is in \( MAX-CUT \).

\[ \leq \]

Given \( \langle G, k \rangle \), we can build a satisfying \( \not= \)-assignment for \( \phi \). This uses the same strategy as above: if all the literal nodes appear in the cut subset, then we set that literal to true.

Notice that our construction disallows "cheating", e.g., coming up with weird cut subsets that circumvent our invariant. This is ensured by the fact that we have a large number of edges coming out of variable nodes and all of them are required to be in the cut for its size to become \( k \).

**NP-completeness**

We have shown that \( MAX-CUT \) is both in NP and NP-hard. Therefore, it is NP-complete.