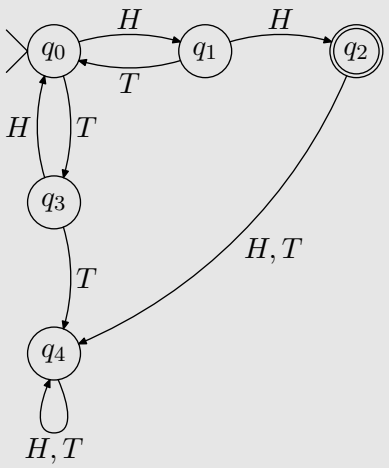
1 More finite deterministic automata

Exercise. *Consider the following game with two players: Repeatedly flip a coin. On heads, player 1 gets a point. On tails, player 2 gets a point. A player wins (and the game ends) as soon as they are ahead by two points. Draw a DFA that recognizes the language of strings (with alphabet $\{H, T\}$) which represent a possible series of flips in which player 1 wins.*

Solution.
Here's a picture of the DFA:



We claim that whenever the DFA is in state q_0, we have a tie so far; whenever the DFA is in state q_1, player 1 is one point ahead; whenever the DFA is in state q_3, player 2 is one point ahead; whenever the DFA is in state q_2, player 2 is two points ahead, and whenever in state q_4, either player 2 is 2 points ahead or the string does not represent a valid sequence of flips (e.g. the game should have ended, but the DFA still has more input to read).

The proof is by induction on the length of the string: it is clearly true for strings of length 1 (see the picture!); assume it's true for a string of length k, and then take any string s of length $k + 1$. Then, it is not difficult to check that, regardless of the last character of s, the transitions work as desired.

Exercise. *Show that a DFA with n states accepts an infinite language if and only if it accepts some string of length at least n.*

Solution.
One direction is clear. For the other, observe that if an n-state DFA accepts a string w of length $m \geq n$, then some state q is visited twice when $w = w_1 \dots w_m$ is processed. Let i, j be such that we first visit q after reading w_i, and we visit q for the second time after reading w_j. Clearly $j > i$, and (by determinism!), $w_1 \dots w_i (w_{i+1} \dots w_j)^k w_{j+1} \dots w_m$ is also in the language. (Pictorially, this means we keep taking the cycle

from q to itself as many times as we want.) Hence there are arbitrarily many words accepted by the DFA.

This is also known as the Pumpkin lemma.[1] Happy Halloween!

2 More Myhill-Nerode

Recall the statement of the Myhill-Nerode theorem from class:

Theorem 1 (Myhill-Nerode)**.**
A language $L \subset \Sigma^*$ is regular if and only if there are only finitely many equivalence classes under the following relation $\sim_L$ on Σ^*:

$$x \sim_L y \iff \forall z \in \Sigma^* : xz \in L \iff yz \in L.$$

Moreover, the minimum number of states in a DFA for L is exactly the number of such equivalence classes.

Proof. ($\implies$) Suppose L is regular. Then there is a DFA M that recognizes L, and for each $x \in \Sigma^*$ there is a state q_x such that on input x, M ends at state q_x. Then, define a relation $\sim_M$ on Σ^* by[2]

$$x \sim_M y \iff q_x = q_y$$

Clearly,

$$x \sim_M y \implies (\forall z \in \Sigma^* : xz \in L \iff yz \in L) \iff x \sim_L y.$$

It follows that each equivalence class of $\sim_M$ is contained entirely in some equivalence class of $\sim_L$, and hence each equivalence class of $\sim_L$ is a union of several equivalence classes of $\sim_M$. Thus, as there are finitely many of the latter, there must be finitely many of the former.

($\impliedby$) In the lecture notes! □

Exercise. *For each of the following languages, determine whether the language is regular or non-regular, and prove your answer:*

1. $\{a^n b a^n \mid n \geq 1\}$.
2. $\{a^{2^n} \mid n \geq 1\}^*$
3. $\{a^n b a^m b a^{m+n} \mid m, n \geq 0\}$

Solution.
1. We apply the Myhill-Nerode theorem: observe that in the infinite set of strings $\{a^n b \mid n \geq 1\}$, no two different ones are equivalent under $\sim_L$, as for $m \neq n$, we have $a^n b a^n \in L$ but $a^m b a^n \notin L$.
2. This is actually regular! Observe that any string in the language must be of the form a^{2k} for some $k \geq 1$, and conversely, $a^{2k} = (a^2)^k$ is in the language.
3. This is essentially saying that DFAs can't do addition (which shouldn't be surprising). We again apply Myhill-Nerode, by observing that the set $\{a^n b a \mid n \geq 0\}$ gives a set of pairwise inequivalent words under $\sim_L$.

[1] Er... did we say pumpkin? We meant pumping :)
[2] Especially if you are unfamiliar with equivalence relations, it is work checking that $\sim_M$ satisfies the conditions for one.

3 Closure properties

On this week's problem set, you will prove that regular languages are closed under homomorphisms and inverse homomorphisms. How might we go about showing closure properties under other natural operations, such as union, intersection, and set difference?

Exercise. *Prove that the complement of a regular language is also regular.*

Solution.
We can take the DFA for the language, and set $F_{\text{complement}} = Q - F_{\text{original}}$. Then, we see that this new DFA accepts a word if and only if the original one did not.

3.1 The product construction.

The product construction is a way to combine two DFAs into a single one that keeps track of both computations simultaneously and independently. Simply, it is a formal implementation of what you would do if you had to run two DFAs on the same string, but you could only pass over the word once (i.e., you're given the kind of access a single DFA would have to the string).

Given DFAs on the same alphabet $M = (Q, \Sigma, \delta, q_0, F), M' = (Q', \Sigma, \delta', q_0', F')$, we define a new DFA $M \times M'$ whose states are $Q_\times = Q \times Q'$, with initial state (q_0, q_0'), and transition function $\delta_\times((q, q'), \sigma) = (\delta(q, \sigma), \delta'(q', \sigma))$ (and let's leave the set of final states unspecified for now).

Then an easy induction shows that

Lemma 1.
On input w, if M, M' are in states q_i, q_i' after reading the first i symbols, $M \times M'$ is in state (q_i, q_i').

Getting the above statement was our initial motivation for coming up with the product construction, and it shows us how to construct DFAs that recognize $L(M) \cup L(M')$ and $L(M) \cap L(M')$.

3.2 Closure under union, intersection, and difference

Exercise. *How does the product construction help us establish closure under union, intersection, and difference?*

Solution.
We already did most of the work in the previous section! Now we just need to set the final states of $M \times M'$ in the appropriate way; for this, we carry over the set operations $\cup, \cap, -$ on an element-by-element basis on the final states of M, M'. So, if we let the final states of $M \times M'$ be

$$F_\times = \{(q, q') \mid q \in F \text{ and } q' \in F'\},$$

it will recognize $L(M) \cap L(M')$; and if we let

$$F_\times = \{(q, q') \mid q \in F \text{ or } q' \in F'\},$$

it will recognize $L(M) \cup L(M')$. Similarly, for the difference $L(M) - L(M')$ we can set

$$F_\times = \{(q, q') \mid q \in F \text{ and } q' \notin F'\}.$$

Alternatively, note that $L(M) - L(M') = L(M) \cap \overline{L(M')}$, where $\overline{L(M')}$ denotes the complement of $L(M')$. Then, because regular languages are closed under intersection and complement, we see that they are also closed under difference.

3.3 Concatenation and Kleene star

When you think about it for a while, for these closure properties, it is more convenient to use the NFA model, because we intuitively want to make ε transitions. Indeed, if you go back to the previous problem, you can find a fairly straightforward NFA construction for the union of two regular languages.

Exercise. *How would you make an NFA that recognizes the concatenation of the languages $L(N), L(N')$ for two NFAs N, N'?*

Solution.
Here's how to obtain the NFA for concatenation: put the two NFAs N, N' side by side, and draw ε transitions from each final state of N to the initial state of N'; let the set of final states be the set of final states of N'.

Clearly, any string in $L(N) \cdot L(N')$ can be accepted; conversely, if a string is accepted, it must take one of the ε transitions we added; this gives us a way to break it into two parts, such that the first part is in $L(N)$ and the second in $L(N')$.

Exercise. *How would you make an NFA that recognizes the Kleene star of the language $L(N)$, for an NFA N?*

Solution.
One idea that comes to mind is to put ε-transitions from each final state of N to the initial state. An issue is that $\varepsilon \in (L(N))^*$, while it might not be true that $\varepsilon \in L(N)$. To fix this, add a new initial state q_s which is also final, and connect it to the initial state of N via an ε transition.

As above, it's easy to see that any string in $L(N)^*$ has an accepting path, and conversely, if there is an accepting path, the occasions on which we use the ε transitions we added give us a way to break the input in parts, each of which is in $L(N)$.

Exercise. *Let L be the language of all strings over the English alphabet that contain exactly one of the strings 'alan', 'mathison', 'turing'. Is L regular?*

Solution.
Yes. There is a way to come up with a DFA for L from first principles, but it's a little messy. Here's a

cleaner solution using closure properties. Let $L_{alan} = \Sigma^* \cdot \{alan\} \cdot \Sigma^*$, i.e. the language of strings that contain 'alan' as a substring. Define $L_{mathison}, L_{turing}$ similarly. Now let

$$L' = \Sigma^* - (L_{alan} \cup L_{mathison} \cup L_{turing})$$

Then L is the language of strings that don't contain any of 'alan', 'mathison', 'turing' as substrings. Then

$$L = (L' \cdot \{alan\} \cdot L') \cup (L' \cdot \{mathison\} \cdot L') \cup (L' \cdot \{turing\} \cdot L')$$

is regular by the closure properties we established above.

4 A word on reductions

Recall:
Definition 2.
We say a language L_1 **polynomial-time mapping reduces** to a language L_2, written as $L_1 \leq_P L_2$, iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that for all $x \in \Sigma_1^*$, $x \in L_1 \iff f(x) \in L_2$.

People get the direction wrong all the time, and that's fine (as long as they get it right in the end!). Here's one way to get it wrong less often: $A \leq_P B$ can be intuitively interpreted by replacing A and B with 'hardness of A' and 'hardness of B'; thus, the 'inequality' means that problem B is *at least* as hard to solve as problem A.

More generally, a mistake people often make when they're new to reductions and are attempting to prove a statement of the form "Show that a problem A is 'hard', for some meaning of 'hard'", is to start by "We reduce A *to* a problem B we know is 'hard' in the desired sense". However, this goes the wrong way; you need to reduce *from* a hard problem.

Reducing *to* a hard problem tells you that if you can solve the hard problem B, that allows you to solve A; yet, A may still be very easy. In terms of the above notation, this corresponds to $A \leq_P B$. Reducing *from* a hard problem means that your original problem allows you to solve the hard problem, and thus must capture its 'hardness' in a sense. In terms of notation, we have $A \geq_P B$.

5 NP-completeness

The motivation behind NP completeness is that we want to capture the 'essence' of the class NP. Recall the two basic definitions:
Definition 3.
A language L is NP**-hard** if it is 'harder than everything in NP'; formally,

$$L \text{ is NP-hard} \iff \forall L' \in \text{NP} : L' \leq_P L$$

Definition 4.
A language L is NP**-complete** if $L \in$ NP and it is NP-hard.

As it turns out, there are many problems that are NP-complete, and that can actually be thought of as 'the same' problem when we ignore polynomial-time differences. Thus, the property of being NP-complete is one way of describing what all these problems are the 'same as'.

6 Examples

Exercise. *(Longest Path) Prove that the* longest path *language from HW4, namely*

$$L = \{\langle G, k\rangle : G \text{ has a simple path of length at least } k\}$$

is NP*-complete. (Hint: reduce from* Hamiltonian Circuit*, which you may assume is* NP*-complete.)*

Solution.
First, it is clear that $L \in$ NP, as we can take the path as the certificate and efficiently verify it. Next, recall that

HamiltonianCircuit $= \{G \mid G$ is an undirected graph with a cycle that touches each vertex exactly once$\}$

is NP-complete by assumption. We first reduce this problem to

HamiltonianPath $= \{G \mid G$ is an undirected graph with a simple path that touches each vertex exactly once $\}$,

that is, we show that
Lemma 2.
HamiltonianCircuit $\leq_P$ HamiltonianPath.

Proof. Given a graph $G = (V, E)$, construct a new graph $G' = (V', E')$ where for some arbitrary vertex $v \in V$ we have $V' = V \cup \{v', s, t\}$ where v', s, t are new vertices, such that v' is connected by an edge to all neighbors of v and to t, and s is connected by an edge to v. Clearly, this is doable in polynomial time.

We claim that G has a Hamiltonian cycle $\iff$ G' has a Hamiltonian path. For the forward direction, we can make a Hamiltonian path in G' by starting as $s \to v$, then tracing the Hamiltonian cycle in G, and when we are supposed to get back to v, we go to v' instead (which is possible by construction). Then we finally take $v' \to t$.

For the reverse direction, given a Hamiltonian path in G', it must start at s and end at t. Then the portion of the path between v and v' can be used to obtain a Hamiltonian cycle in G: simply redirect the edge going into v' to the corresponding edge going into v.

A subtle point is that this will fail to be a cycle if it forces us to take the same edge back and forth. This happens if and only if the Hamiltonian path in G' has only two edges besides $s \to v$ and $v' \to t$, which means that G was a single edge. We can handle this case in our reduction explicitly: if G is a single edge, we let our reduction return two disjoint vertices. □

Now reducing from HamiltonianPath to LongestPath is easy: the instance G of the first problem is equivalent to the instance $\langle G, n-1\rangle$ of the second.

Exercise. *(Bounded-Occurrence SAT) Let*

$$\text{CNF}_k = \{\langle \phi\rangle : \phi \text{ is a satisfiable CNF-formula in which each variable appears at most } k \text{ times}\}.$$

Note that a variable x and its negation $\overline{x}$ count as two occurrences of the same variable, and that CNF_2 *is different from* k-SAT *(the language consisting of satisfiable CNF-formulas where each clause has k literals).*

1. *Prove that* $\text{CNF}_2 \in$ P.

2. Prove that CNF_3 *is* NP*-complete.*

Solution. 1. We use the method of resolution to take the variables out one by one. First, observe that if the variable x appears only as x or only as $\neg x$ in the formula, or if it appears only once, we can satisfy whatever clauses it takes part of and thus exclude x and these clauses. On the other hand, if x and $\neg x$ are both clauses for some variable, we know our formula is not satisfiable. So the interesting case is when we have x in one clause, c_+, and $\neg x$ in another, c_-, such that $(c_+, c_-) \neq (x, \neg x)$. Call the remaining part of the formula r.

We claim that $c_+ \wedge c_- \wedge r$ is satisfiable if and only if $c \wedge r$ is satisfiable, where c is the disjunction of all literals from c_+ and c_- that are not x or $\neg x$. Indeed, given a satisfying assignment of $c_+ \wedge c_- \wedge r$, some literal in either c_+ or c_- has to be satisfied, because we can't have both x and $\neg x$ be satisfied. Thus, forgetting the assignment of x, we get a satisfying assignment of $c \wedge r$. Conversely, given a satisfying assignment of $c \wedge r$, some literal different from x and $\neg x$ will be satisfied in either c_+ or c_-; the other one can be satisfied by assigning x in an appropriate way.

Thus, we have a procedure to decide satisfiability of a CNF_2 formula ϕ efficiently: reduce the number of variables by the methods described above; if there is no contradiction along the way and you get to the empty formula, then ϕ is satisfiable. Otherwise, it is not.

2. We reduce from 3-SAT. Given a 3-CNF formula ϕ, the idea is to give all variables different names, and then add additional clauses that force all aliases of a variable to have the same value.

More specifically, say we had a variable x in ϕ, and we gave all occurrences of x in ϕ different names $x_1, \ldots, x_k$. Then we define a formula

$$\phi_x = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge \ldots \wedge (x_k \vee \neg x_1)$$

The key observation is that $a_1, \ldots, a_k$ is a satisfying assignment of ϕ_x iff $a_1 = \ldots = a_k$. The reverse direction is immediate. For the forward direction, if $a_1 = 1$, then $a_k = 1$ because of the last clause; then $a_{k-1} = 1$ because of the next-to-last clause, and so on. Similarly, if $a_1 = 0$, then $a_2 = 0$ because of the first clause, $a_3 = 1$ because of the second clause, and so on.

Now we form the formula $\varphi = \phi' \wedge \bigwedge_x \phi_x$, where ϕ' is the version of ϕ where all variables have different names, and the conjunction is over all variables x of ϕ. The claim is that ϕ is satisfiable iff φ is.

For the forward direction, given a satisfying assignment of ϕ, we use it to get an assignment of ϕ' in the obvious way; then the ϕ_x will all be satisfied as well, as we assign the same value to each alias of a variable x. For the reverse direction, given a satisfying assignment of φ, all aliases of x must have the same value; thus, the assignment of ϕ' gives us a satisfying assignment of ϕ.

(Oops! I forgot to prove that CNF_3 is in NP! That happens all the time. But my job dictates that I tell you not to forget it, ever! Also, you'll lose points on problem sets if you do. It's another CS-Theory bonding experience, like Turing machines.)

Exercise. *Let* DoubleSAT $= \{\phi \mid \phi$ *is a boolean formula with at least 2 satisfying assignments*$\}$*. Show that* DoubleSAT *is* NP*-complete.*

Solution.
First, clearly DoubleSAT is in NP, as the two assignments can serve as the certificate. Next, we reduce

from SAT. Given a boolean formula ϕ, our reduction returns $\phi' = \phi \wedge (x \vee \neg x)$ for a new variable x. Then, if we have a satisfying assignment of ϕ, we can come up with two different satisfying assignments of ϕ' by additionally setting $x = 0$ or $x = 1$. Conversely, if ϕ is *not* satisfiable, then ϕ' cannot be either!

Exercise. *A subset of the nodes of a graph G is a **dominating set** if every other node in G is adjacent to some node in the subset. Show that*

$$\textsc{DominatingSet} = \{\langle G, k\rangle \mid G \text{ has a dominating set of size } \leq k\}$$

is NP*-complete.*

Solution.
First, clearly DominatingSet is in NP, as we can efficiently verify if a given set is dominating.

For hardness, the idea is to reduce from VertexCover. Given a graph $G = (V, E)$, we form a new graph $G' = (V', E')$ where for each edge (u, v) in E, we add a new vertex w_{uv} that is connected to both u and v, but nothing else.

We claim that G has a vertex cover of size $\leq k$ iff G' has a dominating set of size $\leq k$.

For the forward direction, observe that a vertex cover C of G directly gives us a dominating set in G' over the corresponding vertices: for each edge $(u, v) \in E$, one of u, v has to be in S, and thus w_{uv} will be adjacent to it as needed.

For the reverse direction, a dominating set D in G' will contain at least one of u, v, w_{uv} for each edge $(u, v) \in E$. Now we define a subset $S \subset V$ using D. First, for all $v \in V \cap D$, we add them to S. Then, if for some edge $(u, v) \in E$ we have that only $w_{uv} \in D$, we add one of u or v to S. Clearly, $|S| \leq |D|$, and moreover, S is a vertex cover of G, as for each edge $(u, v) \in E$, at least one of its endpoints is in S.