(1) For each of the following decision problems, say whether its language $L$ is known to belong to NP or to co-NP. In each case, say what the “witness” is, either for membership in $L$ or in $\tilde{L}$. Give a polynomial bound on the size of the witness in terms of the input length, and briefly explain why a given witness can be checked in polynomial time. (The I/O alphabet $\Sigma$ always includes at least two characters. In many cases, “polynomial” can be “linear.” The problems are named after films from the year 2000.)

(a) “Unbreakable”

Instance: An undirected graph $G$ with an even number $n$ of vertices, and a number $k$ such that $0 < k < (n^2 - n)/2$.

Question: Is it possible to break $G$ into two disjoint pieces of $n/2$ nodes each by removing at most $k$ edges?

Answer: “Unbreakable” is in NP. When the answer to an instance $(G, k)$ is “yes,” the witness can be a set of $k$ edges whose removal breaks $G$ into two equal pieces. Such a set has size at most $(n^2 - n)/2$, and the predicate “removing these $k$ edges breaks $G$ into two equal pieces” is decidable in poly-time by doing a breadth-first search from each endpoint of one of those edges, and verifying that each BFS yields $n/2$ vertices.

Logic cue: “does there exist a way to break $G$ by removing...”

(b) “Pay It Forward”

Instance: A nondeterministic Turing machine $N$ with three-way branching at each step, an input $x \in \Sigma^*$ to $N$, and a number $d > |x|$ given in unary notation as the string $0^d$.

Question: Do all possible computation paths by $N(x)$ lead to halting and accepting within $d$ steps?

Answer: “Pay it Forward” is in co-NP. When the answer is no, the witness can be a rejecting computation path. The string $y$ for such a path can be a ternary string of length at most $d$, and since “$0^d$” is part of the input, the length of $y$ is linear in the input length. To verify that $y$ represents a rejecting path, we need to simulate $N$ for $d$ steps using $y$ to resolve nondeterminism. With $N$ given as a list of tuples (or as a .tmt file in the Turing Kit), and with the input $x$ to $N$ being given as part of the instance, we need only appeal to the ability to simulate each move of $N$ by taking one left-to-right “sweep” thru the code of $N$. This takes $O(|N| * d)$ steps to do $d$ steps of $N$, but since the input length $n >= |N| + d$, that’s polynomial in $n$.

(If $N$ has an arbitrarily large number $k$ of tapes, then we can still simulate $N$ through the $k$-tapes-to-one simulation in class. Then we might need $2k$-many left-to-right sweeps thru the code of $N$ to update each head, plus the left-to-right-to-left sweep thru the current contents of the one “real” tape we use to simulate the $k$ “virtual tapes” of $N$. However, over $d$ steps, the contents have size at most $(|x| + d)$ multiplied by the number of bits required to represent a character in $N$’s work alphabet; since this number is certainly less than $|N|$, the contents have size at most $|N|(|x| + d)$, and we get the $d$ steps of $N$ done in time at most $d(|N|(|x| + d) + 2|N|)$, which is polynomial in $|N| + |x| + d$. It was OK not to address this technical point—one can almost always assume that a TM given as input has 2 worktapes with alphabet \{0, 1, B\}.)

Logic cue: ”...all possible computation paths...”
(c) “Traffic”

**Instance:** An airport with \( k \) runways and \( n \) flights that have to land within an \( h \)-hour period. The input specifies for each flight a 30-minute time period when it can land, and which runways it can and cannot land on. No two flights may land on the same runway within fewer than 5 minutes of each other.

**Question:** Is there an air-traffic control schedule that enables all the planes to land?

**Answer:** “Traffic” is in **NP**: Given \( k, n, h \), and most importantly, the **table** of allowable landing times that is part of the instance, the witness is a workable schedule. The schedule has length roughly linear in the number of flights, hence linear in the size of the input table. Verifying that the schedule works likewise takes time roughly proportional to the number of flights, and is an easy matter of checking that all times stated by the schedule are within the stated interval for each flight, and at least 5 min. apart on each individual runway.

Logic cue: “...does there exist a schedule...”

(d) “Hidden Dragon”

**Instance:** A deterministic Turing machine \( M \) whose code includes a “clock” that shuts off computations on inputs of length \( n \) after \( n^2 \) steps, and a string \( y \in \Sigma^* \).

**Question:** Is there a string \( x \) of the same length as \( y \) such that \( M(x) \) outputs \( y \) and halts?

**Answer:** “Hidden Dragon” is in **NP**: When the answer to an instance \((M, y)\) is “yes,” the witness is the string \( x \) of length \(|y|\) such that \( M(x) = y \). The length of \( x \) is (sub-)linear in \(|(M, y)|\), and verification takes at most \(|x|^2 = |y|^2\) steps of \( M \) via the clock. (The extra parenthetical technical remarks in (b) can apply here as well, to multiply things by one or two more factors of \(|M|\) and/or \(|y|\), but that’s still bounded by a polynomial in \(|M| + |y|\), and both here and in (d), the **Hennie-Stearns Theorem** (Theorem 5.6/Corollary 5.5 on p94) knocks down these overhead factors to \( O(\log |M| + \log |y|) \).)

Logic cue: “...does there exist an x...”

(e) “Gladiator”

**Instance:** A ’bot program \( P \) written in C that plays “Player 1” of a two-player game of Quake III Arena against other ’bots that play “Player 2.” Let \( n \) be the size in bytes of the source code for \( P \).

**Question:** Does \( P \) defeat all Player 2 ’bots of the same source-code size, in “Quake” games that last at most \( n \) milliseconds and use no randomness?

**Answer:** “Gladiator” as stated is in **co-NP**: The witness for a no answer is a gamebot program \( Q \) taking Player-2 that beats \( P \). The length of \( Q \) is stated to be the same as that of the input \( P \), hence “linear” in \( n = |P| \). Verification can consist of a single run of \( P \) and \( Q \) on a real computer; since this takes \( n \) milliseconds and a TM suffers at most a quadratic slowdown when simulating a RAM program representing the compiled code, the TM-defined runtime to verify that \( Q \) beats \( P \) is polynomial in \( n \).

Logic cue: “...does \( P \) beat all Player-2 ’bots...”
Let $C = (\ell_1 \lor \ell_2 \lor \ell_3)$ be a clause with three literals, where $\ell_1$ is $x$ or $\bar{x}$, $\ell_2$ is $y$ or $\bar{y}$, and $\ell_3$ is $z$ or $\bar{z}$. (It is often convenient to use this notation so that you can write, e.g., $\bar{\ell}_1$ without having to specify whether $\ell_1$ is $x$ or $\bar{x}$.) Show how to map $C$ to a sub-formula $\psi_C$, where $\psi_C$ is a conjunction of clauses $C_1 \land C_2 \land \cdots \land C_k$ with extra variables $w, v, u, \ldots$, such that:

an assignment $a$ to $x, y, z$ satisfies $C$ if and only if $a$ can be extended to an assignment $a'$ of all the variables that not only satisfies $\psi_C$, but also makes exactly one literal in each of its $k$ clauses true.

Note that if $a$ is the one assignment (out of eight) that fails to satisfy $C$, then you may still be able to satisfy $\psi_C$, but the point is that you won’t be able to make exactly one literal in each of its clauses true. Use this fact to reduce 3SAT to each of the following two problems:

**Exactly One 3SAT**

**Instance:** A Boolean formula $\phi(x_1, \ldots, x_n) = C_1 \land \cdots \land C_m$ in 3CNF.

**Question:** Is there an assignment to the variables that makes exactly one literal in each clause true?

**Exactly Two 3SAT**

**Instance:** A Boolean formula $\phi(x_1, \ldots, x_n) = C_1 \land \cdots \land C_m$ in 3CNF.

**Question:** Is there an assignment to the variables that makes exactly two literals in each clause true? (30 points total)

**Answer:** One thing to realize right away is that $\psi_C$ cannot have two or more of the three given literals in any one clause. The reason is that at least one of the satisfying assignments to $C$ would automatically make (the) two literals true in that clause, thus preventing the desired correspondence. To be more precise: if the clause is $(\bar{\ell}_1 \lor \bar{\ell}_2 \lor u)$ where $u$ is some new variable, then you could reason that the assignment making both $\ell_1$ and $\ell_2$ false was one that you wanted to exclude anyway, but the problem is that doing this with $\ell_3 = true$ still breaks the correspondence. Thus we need to have the literals in separate clauses and use several other variables along with them. This suggests two patterns to begin with:

$$\psi_C = (\ell_1 \lor u \lor v) \land (\ell_2 \lor w \lor \bar{u}) \land (\ell_3 \lor \bar{w} \lor \bar{v}) \land \cdots$$

or

$$\psi_C = (\bar{\ell}_1 \lor u \lor v) \land (\bar{\ell}_2 \lor w \lor \bar{u}) \land (\bar{\ell}_3 \lor \bar{w} \lor \bar{v}) \land \cdots$$

Whether we want to negate the literals depends on which case we want to be impossible: having to make one of the other two variables in each clause true, or having to make the variables in the other clauses all be false. The latter seems much more extreme, especially if we make the other variables all be different:

$$\psi_C = (\bar{\ell}_1 \lor u \lor v) \land (\bar{\ell}_2 \lor w \lor x) \land (\bar{\ell}_3 \lor y \lor z) \land \cdots$$

Now we need to add some more clauses involving the other variables—maybe involving the original three literals again but it would be nice to keep it simple. After this strategic start, it happens that all you need to do is take one of the new variables from each clause to make one more clause:

$$\psi_C = (\bar{\ell}_1 \lor u \lor v) \land (\bar{\ell}_2 \lor w \lor x) \land (\bar{\ell}_3 \lor y \lor z) \land (u \lor w \lor y).$$

Here is the analysis:
• If the literals are all false, then their negations are true in the first three clauses, which forces the new variables \( l_l \) to be false. But then the fourth clause, \((u \lor w \lor y)\), is unsatisfied. So it is not possible to make exactly one literal true in each clause—which is what we want for the assignment that does not satisfy \( C \).

• If some literal in \( C \) is true, then wlog. by symmetry, suppose it is \( \ell_1 \). Then \( \bar{\ell}_1 \) is false. Then we can take \( u \) to be true and \( v, w, y \) all false. Then we just take \( x \) and \( z \) to be the opposite of whatever \( \ell_2 \) and \( \ell_3 \) are assigned, respectively, and we’ve made exactly one literal in each clause true.

Thus \( \psi_C \) is correct. If we do this to every clause \( C_j \) in a given strict-3CNF formula \( \phi \), then we get a strict-3CNF formula \( \phi' \) such that \( \phi \) is satisfiable if and only if \( \phi' \) has an assignment that satisfies exactly one literal in each clause. I.e.,

\[
\phi \in 3\text{SAT} \iff \phi' \in \text{Exactly One 3SAT}.
\]

Finally, create \( \phi'' \) by negating every literal in \( \phi' \). By the strictness, an assignment that satisfies exactly one literal in each clause of \( \phi' \) equates to one that satisfies exactly two literals in every clause of \( \phi'' \). Thus we get a reduction from 3SAT to Exactly Two 3SAT, so it also is NP-complete.

For a footnote, note that if the original three literals are all negative, then their negations are positive and every variable in \( \psi_C \) is positive. If they are all positive, then we can do the above but negate all of \( u, v, w, x, y, z \)—this doesn’t matter for the analysis. Thus if \( \phi \) is a strict-3CNF formula in which every clause has variables of the same sign, then we can get the same property in \( \phi' \) and \( \phi'' \). This also might be useful…

**Added:** There is in fact a smaller answer by negating two of the literals:

\[
\psi'_C = (\ell_1 \lor w \lor y) \land (\bar{\ell}_2 \lor w \lor x) \land (\bar{\ell}_3 \lor y \lor z).
\]

If \( \ell_1 \) is false then we must make \( w \) or \( y \) true, but either conflicts with \( \ell_2 \) and \( \ell_3 \) both being false, so the all-false assignment fails in \( \psi'_C \). But if one or both of \( \ell_2 \) and \( \ell_3 \) are true then it can work. And if \( \ell_1 \) is true then \( w \) and \( y \) must both be false, which sets up the situation where \( x \) and \( z \) are completely free vis-à-vis \( \ell_2 \) and \( \ell_3 \). The only knock against this solution is that it does not preserve the equal-sign property, but that wasn’t stated in the problem.

(3) Given an undirected graph \( G = (V, E) \), define its line graph \( G' = (V', E') \) by \( V' = E \) and

\[
E' = \{(u, v), (v, w)) : (u, v) \in E \land (v, w) \in E\},
\]

recalling that in an undirected graph, \((u, v) \in E \iff (v, u) \in E\). That is, \( G' \) has a vertex for every edge of \( G \), and two edges that “touch” in \( G \) become an edge in \( G' \). (Example snipped in this key.) Now consider the following decision problem:

**Edge Cover (EC)**

**Instance:** An undirected graph \( G \) and an integer \( k \geq 1 \).

**Question:** Does there exist a set \( H \) of at most \( k \) edges such that every other edge in \( G \) touches an edge in \( H \)?
(a) Does \( f(G, k) = (G', k) \) give a polynomial-time mapping reduction (i) from VERTEX COVER to EDGE COVER?, (ii) from EDGE COVER to VERTEX COVER? (iii) both? (iv) neither? Justify your answer. (12 pts.)

(b) Show that EDGE COVER is NP-complete by mapping reduction from 3SAT. You are allowed to use extra properties of the reduction in the proof of the Cook-Levin theorem or the reduction in problem (2), but you may not need to do so. (30 pts., making 42 on the problem and 102 on the set)

\[ \text{Answer: (a) The answer is neither-nor. The easiest way to see this is the positive fact that the line-graph function } f \text{ gives a mapping reduction from EDGE COVER to the DOMINATING SET problem covered in lecture, not to EDGE COVER. This is because } f \text{ makes edges of } G \text{ become nodes of } G' = f(G) \text{ so that edge-touching-edge in } G \text{ becomes node-adjacent-to-node in } G'. \]

\[ \text{(b) In full formality, where } G = (V, E) \text{ is understood, we have:} \]

\[ L_{EC} = \{ G\#k : (\exists H \subseteq E)[|H| \leq k \land (\forall (u, v) \in E)(\exists (t, w) \in H)[u = t \lor v = t]] \}. \]

Now only the initial \((\exists H \subseteq E)\) is a “heavy” quantifier—the other two are over the polynomially-many edges in the graph. Hence this is an NP-definition of \( L_{EC} \).

For hardness under poly-time many-reductions, we reduce 3SAT to \( L_{EC} \). Given any 3CNF formula \( \phi = C_1 \land \cdots \land C_m \) with \( n \) variables, we build the graph \( f(\phi) = G_\phi \) as follows: We allocate the usual “rung” vertices \( x_i, \bar{x}_i \) and “clause triangles” as before, but we also allocate some extra vertices. Between every pair \( x_i, \bar{x}_i \) we insert a node \( u_i \) to make the edges become \((x_i, u_i)\) and \((u_i, \bar{x}_i)\). To “make assurance double sure” as Macbeth says in Shakespeare’s Scottish play, we also allocate nodes \( v_i \) connected only to \( u_i \). This makes totally clear that to cover the edge \((u_i, v_i)\), one will need to choose one of the two edges \((x_i, u_i)\) or \((u_i, \bar{x}_i)\), since there is no way to cover it from outside. Thus the choices of these edges are in 1-to-1 correspondence with truth assignments. To cover the clause gadgets we need at least one edge per triangle. Hence \( k := n + m \) is the absolute minimum possible size for any edge cover. The “crossing edges” go from any \( x_i \) in \( C_j \) to \( x_i \) in the rung, and from any \( \bar{x}_i \) in \( C_j \) to \( \bar{x}_i \) in the rung. This completes the construction, and its time complexity is clearly polynomial in \( n \) and \( m \).

For correctness, we claim the target \( k \) is attainable iff \( \phi \) is satisfiable. First, if \( a \) is a satisfying assignment to \( \phi \), we can determine the corresponding minimum-size edge cover \( F_a \) by first including the rung edges corresponding to the assignment. Since \( a \) satisfies \( \phi \), ever clause has at least one of its crossing edges already covered. Hence choosing the edge in the triangle that is opposite that crossing edge covers the other two crossing edges, as well as the other two edges within the triangle. Hence our \( F_a \) of size \( n + m \) covers all edges.

Going the other way, suppose we have an edge cover \( F \) of size \( m + n \). Now a fine point is that it is possible to cover a clause triangle without choosing one of its internal edges, by choosing two crossing edges into it instead. However, doing so “wastes” an edge, and makes it impossible to complete an edge cover of the minimum possible size \( m + n \) when you consider the rungs and the other clauses. Indeed, just choosing one crossing edge wastes it. Hence \( F \) must consist solely of \( m \) edges within the \( m \) clause triangles (one in each) and \( n \) in the rungs. The only way a single triangle edge in a clause gadget can work is if the crossing edge from the opposite vertex of the triangle is covered by the rung it goes to. This means that the corresponding clause is satisfied by the truth assignment from the rungs. Since this happens for each clause, \( \phi \) must be satisfied by the truth assignment made in the rungs. Hence the reduction is correct.